

OGRE

Adkoba & Aklal

November 30, 2014

Contents

I Bases de Ogre	11
1 Installation, Premier programme	13
1.1 Premier pas	13
1.1.1 Sources	13
1.1.2 Installation	13
1.2 Premiere compilation	13
1.2.1 Premier programme	13
1.2.2 Compilation	16
1.3 Premiere compilation sous CodeBlocks	16
1.4 Premiere compilation sous kDevelop	16
1.5 Code de base	16
1.5.1 Source	16
2 Insérer des objets dans la scène	17
2.1 les entités	17
2.1.1 Une entité c'est quoi	17
2.1.2 Le sceneManager	17
2.1.3 Créer une entité	18
2.2 Les Nœuds de la scène	18
2.2.1 L'utilité des Nœuds	18
2.2.2 Créer un nœud	18
2.2.3 Code	19
2.3 Créer un mesh	20
2.3.1 Le mesh	20
2.3.2 Le matériau	22
2.3.3 Code	22
3 Se repérer dans l'espace	25
3.1 Le système de coordonnées	25
3.1.1 Le repère "main droite"	25
3.1.2 Pourquoi main droite ?	25
3.1.3 Repère local, repère absolu	25
3.1.4 Yaw, pitch, roll	26
3.2 Déplacer des objets	26
3.2.1 Bouger un noeud de scène	26
3.2.2 setPosition()	27
3.2.3 translate()	27
3.2.4 Concrètement, ça veut dire quoi ?	28

3.3	La caméra	28
3.3.1	Création	28
3.3.2	Placement	29
3.3.3	Code	29
3.4	Le viewport	32
3.4.1	Plusieurs viewports	33
4	La lumières	37
4.1	Les lumières	37
4.1.1	Quelques fonctions de base	37
4.1.2	Les types de lumières	38
4.1.3	Lumière ponctuelle	38
4.1.4	Lumière directionnelle	39
4.1.5	Projecteur	39
4.2	Les ombres	40
4.2.1	Activer les ombres	40
4.2.2	Les différents types d'ombres	40
4.2.3	Code	41
5	La gestion des entrées	47
5.1	Les frame listeners	47
5.1.1	Des "écouteurs d'images" ?	47
5.1.2	Le frame listener en pratique	48
5.2	OIS	50
5.3	Allons-y sans buffer	51
5.3.1	Explications	51
5.3.2	Création du frame listener	52
5.3.3	Déplacer la caméra	52
5.3.4	Et avec un noeud de scène ?	54
5.3.5	55
5.3.6	Code	55
5.4	Avec buffer, c'est plus simple ?	64
5.4.1	Mise en place	64
5.4.2	Code	70
6	Gardez les pieds sur Terre	77
6.1	Créer un terrain	77
6.1.1	Préparation	77
6.1.2	Quelques paramètres à régler	78
6.1.3	Le terrain	80
6.2	Le plaquage de textures	84
6.2.1	Au boulot !	88
6.2.2	Pour aller plus loin	95
6.3	Les groupes de terrains	99
6.3.1	Création	99
6.3.2	Optimiser le temps de chargement	103
Index		105

Listings

2.1	PremiereApplication.cpp: Instanciation d'entité	20
2.2	PremiereApplication.cpp: Création d'un sol	22
3.1	PremiereApplication.cpp: Création de la caméra	29
3.2	PremiereApplication.h: Création de la caméra	31
3.3	main.cpp: Création de la caméra	31
3.4	Création d'un viewport	33
3.5	Prototype de addViewport	33
3.6	createViewports: création de plusieurs vues	34
3.7	createViewports: création de plusieurs vues	35
4.1	PremiereApplication.h: ajout d'une méthode pour la gestion de lumière et des ombres	41
4.2	PremiereApplication.cpp: ajout d'une méthode pour la gestion de lumière et des ombres	42
5.1	InputListener.h	49
5.2	InputListener.cpp	49
5.3	Include OIS	51
5.4	PremiereApplication::createFrameListener	52
5.5	Capture de l'état ponctuel d'une touche	55
5.6	InputListener.h	55
5.7	InputListener.cpp	56
5.8	PremiereApplication.h	58
5.9	PremiereApplication.cpp	59
5.10	CMakeLists.txt	63
5.11	Activation du buffer pour la souris et le clavier	64
5.12	Classes mères pour gestion des Listener	65
5.13	Enregistrement des listener	65
5.14	Attributs d'InputListener	65
5.15	Constructeur d'InputListener	66
5.16	Méthodes virtuelles appelées lors d'un évènement sur un périphérique	66
5.17	Implémentation de l'appuie sur ECHAP	67
5.18	Implémentation de l'appuie sur les touches de déplacement	67
5.19	Implémentation du déplacement de la caméra dans la méthode frameRenderingQueued	69
5.20	Implémentation de la rotation de la caméra dans la méthode mouseMoved	69
5.21	PremiereApplication::createFrameListener()	70
5.22	InputListener.h	70
5.23	InputListener.cpp	72
6.1	Ajout du fichier d'entête pour la gestion des terrains	77

6.2	Modification de CMakeLists.txt pour l'inclusion des lib Terrain et Paging	77
6.3	Attributs pour la gestion de terrain	78
6.4	Réglage de la caméra	78
6.5	Vérification et réglages de vue à l'infini	78
6.6	Définition de l'éclairage pour le terrain	79
6.7	Méthode pour la prise en charge de l'initialisation du terrain . .	79
6.8	Application des réglages	80
6.9	Création du terrain	80
6.10	Chargement du fichier heightmap	81
6.11	Création de l'objet ImportData pour la définition des paramètres à fournir au terrain	81
6.12	Mise en place d'une texture pour le terrain	83
6.13	Choix d'un filtrage anisotrope	84
6.14	Préparation et chargement du terrain	84
6.15	Libération de place en mémoire	84
6.16	Ajout de calques	84
6.17	Récupération du Blend Map pour le premier terrain	85
6.18	Attribution de la transparence désirée sur tous les points du calque	86
6.19	Mise à jour de la Blend Map	86
6.20	createTerrain (code complet)	86
6.21	méthode getHeightAtTerrainPosition pour trouver l'altitude du terrain en fonction de la position	88
6.22	Méthode convertImageToTerrainSpace pour convertir les coordonnées de l'image en coordonnées du terrain	88
6.23	Attribution de la transparence désirée sur tous les points du calque selon leur position	88
6.24	createTerrain	89
6.25	PremiereApplication.h	92
6.26	CMakeLists.txt	93
6.27	Récupération des Blend Map pour le premier et le second terrain	95
6.28	Utilisation de la méthode statique Clamp()	96
6.29	Plaquage de textures sur zone	97
6.30	TerrainGroup: include et création	99
6.31	Création des terrains du groupe	100
6.32	PremiereApplication.definirTerrain	101

Avant-Propos

Présentation du document

This document is a re-lecture of the tutorial "Découvrez Ogre 3D". The original tutorial was written by Julien Chichignoud on Openclassroom (aka "Le site des zéros").

Writing this document is an attempt to make something like an active reading, and thus have a more attentive lecture and deepest understanding of Ogre's concepts. We modified a lot (sentences, structure, ...) but the essence is the same as the original tutorial.

We wish to highlight the fact that this document is a personal interpretation, and a in progress work.

The original tutorial is in French and as we are spending most of our strength in the comprehension of Ogre we did not take time to translate the original tutorial in French.

Licence

The original tutorial is released under the Creative Common licence and so is also released this document.

The rest of the document will be now in French, sorry for those who cannot take advantage of the document.

Part I

Bases de Ogre

Chapter 1

Installation, Premier programme

1.1 Premier pas

1.1.1 Sources

J'ai d'abord voulu suivre le tutoriel du site du zéro <http://fr.openclassrooms.com/informatique/cours/decouvrez-ogre-3d/>

Mais ce tutoriel du site des zéros semble plus axé visual studio, par contre un lien est donné pour la compilation de projets sous Ubuntu <http://geenux.wordpress.com/2010/03/18/installation-de-ogre-1-7-et-compilation-avec-cmake-sous-ubuntu/>

1.1.2 Installation

Installation avec synaptic des packages suivants:

1. libogre-dev
2. ogre-samples
3. ogre-samples-data
4. libogre-1.7.4
5. liboios-1.3.0
6. liboios-dev

1.2 Première compilation

1.2.1 Premier programme

- Création d'un répertoire contenant les fichiers suivants:

```
Hraesvelg:~/Documents/workspace/3D/OGRE/←
  ExampleApplication> ls -l
total 24
-rw-rw-r-- 1 oliver oliver 1109 fvr. 23 12:16 ←
  CMakeLists.txt
-rw-rw-r-- 1 oliver oliver  844 fvr. 23 12:02 ←
  helloworld.cpp
-rw-rw-r-- 1 oliver oliver 4894 fvr. 23 12:18 Makefile
-rw-r--r-- 1 oliver oliver  446 fvr. 23 12:20 plugins.←
  cfg
-rw-r--r-- 1 oliver oliver 1391 fvr. 23 12:20 resources←
  .cfg
```

- helloworld.cpp:

```
\#include "ExampleApplication.h"

class TutorialApplication: public ExampleApplication
{
protected:
public:
    TutorialApplication()
    {
    }

    ~TutorialApplication()
    {
    }
protected:
    void createscene(void)
    {
    }
};

#if OGREPLATFORM == OGRE_PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
#include "windows.h"

INT WINAPI WinMain( HINSTANCE hInst , HINSTANCE, LPSTR ←
strCmdLine , INT )
#else
int main(int argc, char **argv)
#endif
{
    // Create application object
    TutorialApplication app;

    try {
```

```

        app.go();
    } catch( Exception& e ) {
#ifndef OGREPLATFORM == OGRE_PLATFORM_WIN32
    MessageBox( NULL, e.what(), "An exception has occurred!", MB_OK | MB_ICONERROR | MB_TASKMODAL );
#else
    fprintf(stderr, "An exception has occurred: %s\n",
            e.what());
#endif
}
return 0;
}

```

- CMakeLists.txt est une copie modifiée du CMakeLists.txt donnée dans le tutoriel de compilation sous Ubuntu. Les lignes commençant par `#` sont les lignes originales que j'ai du modifier.

```

project(helloworld)
cmake_minimum_required(VERSION 2.6)

# set(CMAKE_MODULE_PATH "/usr/lib/OGRE/cmake/")
set(CMAKE_MODULE_PATH "/usr/share/OGRE/cmake/modules")
#set(CMAKE_CXX_FLAGS "-Wall -Werror -ansi -pedantic -g")

# Il s'agit du tutoriel d'exemple, qui utilise quelques fichiers prédefini de Ogre. Il faut indiquer à cmake où se trouvent les includes en question
# include_directories (" /usr/share/OGRE/Samples/Common/include/")
include_directories (" /usr/share/OGRE-1.7.4/Samples/Common/include/")

# Bien sûr, pour compiler Ogre, il faut le chercher, et définir le répertoire contenant les includes.
find_package(OGRE REQUIRED)
include_directories (${OGRE_INCLUDE_DIRS})

# L'exemple dépend aussi de OIS, une lib pour gérer la souris, clavier, joystick...
find_package(OIS REQUIRED)

# On définit les sources qu'on veut compiler

```

```

SET(SOURCES
helloworld.cpp)

# On les compile
add_executable(
helloworld ${SOURCES}
)

# Et pour finir, on lie l'executable avec les ↵
librairies que find_package nous a gentillement ↵
trouve.
target_link_libraries(helloworld ${OGRE_LIBRARY} ${←
OIS_LIBRARY})
%
```

plugins.cfg et resources.cfg sont copiés de /usr/share/OGRE-1.7.4

1.2.2 Compilation

les commandes à faire sont:

- cmake.
- make
- ./helloworld

1.3 Première compilation sous CodeBlocks

A tenter!!!!

<http://fr.openclassrooms.com/informatique/cours/decouvrez-ogre-3d/configuration-d-un-projet-3>

1.4 Première compilation sous kDevelop

J'ai suivi

<http://www.ogre3d.org/tikiwiki/tiki-index.php?page=Setting+Up+An+Application+-+KDevelop+-+Linux>.

Je copie dans le répertoire ogretest le répertoire d'exemple Sample_Water. Le processus foire.

1.5 Code de base

1.5.1 Source

<http://fr.openclassrooms.com/informatique/cours/decouvrez-ogre-3d/le-code-de-base-1>

Ce code de base est en fait le même que celui fait dans le chapitre Première Compilation, sauf qu'alors on avait que un seul fichier cpp, on rajoute ici un fichier.h qui est inclus dans le fichier main.cpp.

Chapter 2

Insérer des objets dans la scène

2.1 les entités

2.1.1 Une entité c'est quoi

Une entité est l'ensemble des informations attachées aux polygones constituant un objet 3D.

Un Mesh étant l'ensemble des polygones constituant un objet 3D, une entité est l'ensemble des informations attachées à un Mesh:

- les vertices (sommets des polygones),
- les textures,
- un squelette si le modèle est sujet à des animations
- ...

Tous les objets solides qui apparaissent à l'écran sont donc des entités et sont représentés par une seule et même classe dans Ogre: la classe Entity.

2.1.2 Le sceneManager

C'est au sceneManager que revient la tâche de créer tous les objets que peut contenir la scène (tous les modèles, lumières, caméra et autres objets) et de nous permettre ensuite d'y accéder.

Par conséquent, l'insertion d'objets dans la scène passe toujours par lui (ou par l'un des éléments déjà insérés) et par les méthodes qu'il propose pour cela. Il en existe diverses variantes en fonction de la scène que l'on veut réaliser; selon que celle-ci sera par exemple en intérieur ou en extérieur, par exemple.

Toutes les applications Ogre doivent donc avoir un sceneManager pour pouvoir fonctionner, puisque c'est lui qui s'occupe de tout ! La classe ExampleApplication ne fait pas exception et possède donc un attribut msceneMgr, qui est un pointeur sur le sceneManager de l'application et qui nous permettra dans les parties suivantes d'agrémenter notre scène avec des objets.

2.1.3 Créeer une entité

L'entité doit être créée par le sceneManager pour pouvoir être ajoutée à la scène. La méthode createEntity() permet de faire cela.

```
Entity *head= msceneMgr->createEntity( "Tete" , "←  
ogrehead.mesh" );
```

1

- Le premier paramètre est le nom que vous souhaitez donner au mesh. Ce nom doit être unique!
- Le second paramètre est le nom du fichier que vous voulez charger. Notez l'extension.mesh, qui est le format de fichiers pour les modèles reconnus par Ogre.

Le fichier spécifié (avec le deuxième paramètre) se trouve, accompagné d'autres modèles d'exemples, dans le dossier OgreSDK/media/models, qui doit être correctement renseigné dans le fichier resources.cfg pour qu'Ogre puisse le trouver lors de l'exécution.²

Le mesh a été ajouté à la scène mais il nous manque encore une chose avant de pouvoir l'afficher à l'écran...

2.2 Les Nœuds de la scène

2.2.1 L'utilité des Nœuds

Dans Ogre, lorsque l'on souhaite manipuler une entité (un personnage, une lumière, une caméra...), les déplacements que l'on veut effectuer se font par l'intermédiaire d'un nœud de scène, ou *sceneNode*.

Un sceneNode est un objet invisible auquel on va pouvoir attacher un nombre indéfini d'entités, lesquelles deviennent solidaires de ce nœud et subissent donc les mêmes transformations que lui. C'est donc une sorte de conteneur qui contient les informations de positionnement de chacune des entités de la scène qui lui sont rattachées.

Bien sûr on pourrait déplacer nos entités directement mais avec un nœud on pourra déplacer en une fois toutes les entités attachées au nœud.

Quoi qu'il en soit, attacher chaque entité à un nœud est primordial, sans quoi elle ne s'affichera pas dans votre scène!

2.2.2 Créeer un nœud

Pour créer un nœud de scène on devra passer par un nœud déjà existant, ce qui va nous permettre d'avoir des relations d'héritage entre nos noeuds.

On utilisera une méthode dédiée:

¹Pourquoi utilise t-on -t pour appeler une méthode d'une classe?

²Mais à quoi sert ce fichier .mesh?

```
sceneNode *noeudEnfant = noeudParent->↔
    createChildSceneNode( "enfant" , Vector3::ZERO, ←
        Quaternion::IDENTITY) ;
```

Mais comment fait-on pour le premier noeud qu'on va créer ? Le noeud "racine" existe dès que le sceneManager est créé, c'est un noeud comme un autre avec les même³ méthodes mais ce noeud est unique.

Nous récupérons ce noeud racine de la scène à l'aide de l'instance du scene-Manager:

```
sceneNode *node= msceneMgr->getRootSceneNode()->↔
    createChildSceneNode( "nodeTete" , Vector3::ZERO, ←
        Quaternion::IDENTITY) ;
```

La méthode getRootSceneNode() nous permet de récupérer un pointeur sur le noeud racine unique de la scène. On appelle ensuite sa méthode createChildSceneNode pour lui ajouter un nouveau noeud fils.

Notez qu'aucun des paramètres de la méthode n'est obligatoire, pour information:

- Le premier argument est le nom que vous voulez donner à votre noeud. Sur le même principe que les entités ce nom pourra être utilisé pour récupérer un pointeur vers le noeud en question.
- Le deuxième argument est la position initiale du noeud.
- Le troisième argument est le quaternion avec lequel vous voulez initialiser votre noeud.

Sachez pour le moment qu'un quaternion est un objet mathématique qui permet de faire faire aux objets des rotations dans l'espace⁴. Quaternion::IDENTITY⁵ lui dit de ne pas faire de rotation.

Avec ce code en main, vous pouvez simplement attacher l'entité précédemment créée au noeud avec la ligne suivante:

```
node->attachObject ( head ) ;
```

En compilant, vous devriez voir la tête d'un ogre au milieu de l'écran. Vous pouvez déplacer la caméra avec Z, S, Q, D et la souris pour voir ce que ça donne de plus près.

2.2.3 Code

PremiereApplication.cpp

³"les même" faut il un "s" à "même" ?

⁴Que des rotations?

⁵comment écrire Quaternion::IDENTITY de manière élégante?

Listing 2.1: PremiereApplication.cpp: Instanciation d'entité

```
#include "PremiereApplication.h"

void PremiereApplication::createScene()
{
    //creation d une entite
    Entity *head= mSceneMgr->createEntity("Tete", "←
        ogrehead.mesh" );

    //creation d un noeud
    SceneNode *node= mSceneMgr->getRootSceneNode( )->←
        createChildSceneNode( "nodeTete" , Vector3::←
            ZERO, Quaternion::IDENTITY);

    node->yaw( Radian(Math::PI));
    node->yaw( Radian(Math::PI));

    Vector3 position = Vector3(30.0, 50.0, 0.0);
    node->setPosition( position);

    node->setPosition(30.0, 50.0, 0.0);
    node->translate(-30.0, 50.0, 0.0);

    //attachement de l entite au noeud
    node->attachObject( head );
}
```

2.3 Créer un mesh

Nous pouvons rajouter un sol à notre scène, pour cela nous allons créer nous-mêmes un nouveau mesh. étant donné que nous n'avons besoin que d'un plan pour le sol, le mesh peut très simplement être créé dans le code de notre application.

2.3.1 Le mesh

Il existe une classe Plane⁶ qui va nous permettre de générer... un plan qui représentera le sol.⁷

Pour créer un plan nous appelons Plane avec les paramètres suivants:

- le premier paramètre permet de définir le vecteur normal au plan à créer (ici l'axe Y pour que notre plan soit horizontal).

⁶de même que pour quaternion.entity marquer les noms de classe de manière élégante pour les discerner du texte serait bien

⁷En plus de la classe Plane, vous trouverez aussi des classes Box (pour les cubes) et Sphere qui fonctionnent sur le même principe.

- le second paramètre est la distance à l'origine de la scène dans le sens du vecteur normal (ici, je mets 0 pour que mon mesh plan soit centré).

```
Plane plan (Vector3::UNIT_Y, 0);
```

Une fois le plan créé, il faut que l'on crée un mesh, c'est-à-dire l'objet 3D en lui-même (la représentation du plan) qui sera visible dans la scène. Pour cela, on utilise le Mesh Manager, qui va s'occuper de créer les faces de notre mesh.

```
MeshManager::getSingleton().createPlane( "sol", ←
    ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, ←
    plan, 500, 500, 1, 1, true, 1, 1, 1, Vector3::UNIT_Z←
);
```

Quelques explications sur cette ligne s'imposent.

- Tout d'abord, la méthode statique `getSingleton()` permet de récupérer un objet instancié de façon unique, donc ici notre `MeshManager`.
- Les deux premiers paramètres correspondent respectivement au nom que l'on veut donner à notre mesh et au nom du groupe auquel on veut qu'il appartienne.
- Suivent ensuite le nom du plan à modéliser, puis la largeur et la hauteur qu'il doit avoir, puis le nombre de subdivisions du plan dans ces deux sens. Plus il y a de subdivisions, plus il y a de polygones dans notre mesh.
- Le booléen suivant indique que les normales sont perpendiculaires au plan.
- Les trois paramètres suivants sont le nombre de textures que l'on va pouvoir assigner au plan, puis le nombre de fois que la texture sera répétée dans les deux directions.
- le dernier paramètre est le vecteur indiquant la direction du haut du mesh. Attention: il ne faut pas le confondre avec la normale du plan, qui est différente.

Il reste encore des paramètres par défaut que l'on verra plus tard.

Enfin, nous allons revenir vers un code connu: nous allons créer l'entité qui représentera le plan. C'est le même principe que tout à l'heure:

- tout d'abord, on crée une entité à partir du scène Manager en la nommant et en lui indiquant le mesh à utiliser.
- On crée ensuite un nouveau nœud à partir du nœud racine et on l'attache à notre entité.

```
//creation d'une entité
Entity *ent= mssceneMgr->createEntity( "EntiteSol", ←
    "sol" ); ←

//creation d'un nouveau noeud
node = mssceneMgr->getRootSceneNode()->←
    createChildSceneNode();
```

```
//on attache le noeud a notre entite
node->attachObject(ent);
```

2.3.2 Le matériau

Nous allons finir en ajoutant une texture au sol: de l'herbe. Pour cela, il suffit de rajouter la ligne suivante après la création de l'entité:

```
ent->setMaterialName( "Examples/GrassFloor" );
```

Si vous voulez connaître les matériaux fournis avec Ogre, il vous suffit d'aller dans le dossier media/materials/scripts. Ici, on prend le matériau GrassFloor enregistré dans le fichier Examples.material.

Les textures correspondantes se trouvent dans le dossier media/materials/-textures, si vous voulez faire des essais.

Vous pouvez maintenant exécuter votre programme.

Lancez l'application et remontez la caméra avec la souris et les touches de déplacement, vous devriez voir quelque chose ressemblant à la capture suivante.

Image utilisateur

Euh... La tête d'Ogre est coupée par le sol en herbe...

En effet, notre plan est centré sur l'origine de la scène, et l'on a aussi placé notre tête à l'origine. Mais quelle partie de la tête est à l'altitude 0 ?

Ici, c'est donc un point au milieu de la tête, puisque le plan d'herbe passe par là. Cependant, ce point n'est pas nécessairement au milieu de l'objet que vous intégrez. Cela dépend de la personne qui a modélisé l'objet et qui a donc décidé par rapport à quel point on allait définir la position du mesh. Pour un personnage, on pourrait mettre ce point à ses pieds, pour que l'altitude 0 corresponde effectivement au moment où le personnage touche le sol avec ses pieds.

Pour corriger cela, il va falloir remonter notre noeud lié à notre entité. C'est l'objet du prochain chapitre.

2.3.3 Code

PremiereApplication.cpp

Listing 2.2: PremiereApplication.cpp: Création d'un sol

```
#include "PremiereApplication.h"

void PremiereApplication::createScene()
{
    //creation d une entite
    Entity *head= mSceneMgr->createEntity("Tete", "→
        ogrehead.mesh" );

    //creation d un noeud
    SceneNode *node= mSceneMgr->getRootSceneNode( )->←
        createChildSceneNode( "nodeTete" , Vector3::←
            ZERO, Quaternion::IDENTITY);
```

```

node->yaw(Radian(Math::PI));
node->yaw(Radian(Math::PI));

// setPosition place le noeud aux coord passees en ↵
   parametres
Vector3 position = Vector3(30.0, 50.0, 0.0);
node->setPosition(position);

node->setPosition(30.0, 50.0, 0.0);

Vector3 position = Vector3(30.0, 50.0, 0.0);
node->setPosition(position);
*/

//deplace le noeud par rapport a sa position ↵
   actuelle
node->translate(-30.0, 50.0, 0.0); //par defaut la ↵
   trnslt se fait par rap a TS_WORLD

//attachement de l entite au noeud
node->attachObject( head );

//creation d un plan
Plane plan(Vector3::UNIT_Y, 0);

//creation d un mesh cad l objet 3d visible ds la ↵
   scene
MeshManager::getSingleton().createPlane("sol",
    ResourceGroupManager::←
        DEFAULT_RESOURCE_GROUP_NAME,
    plan, 500, 500, 1, 1, true, 1, 1, 1, ←
        Vector3::UNIT_Z);

//entite qui representera le plan
Entity *ent= mSceneMgr->createEntity("EntiteSol", "←
sol");

//ajout du materiau a l entite
ent->setMaterialName("Examples/GrassFloor"); //←
   texture de pelouse

ent->setMaterialName("Examples/WaterStream"); //←
   texture d eau animee*/

//creation d un noeud
node = mSceneMgr->getRootSceneNode()->←
   createChildSceneNode();
node->attachObject(ent);

```

{}

À cause de la position de la caméra, il se peut alors que le sol ne soit pas visible.

Chapter 3

Se repérer dans l'espace

3.1 Le système de coordonnées

3.1.1 Le repère "main droite"

Le repère est orthogonal et orthonormé (les axes/les vecteurs directeurs de ces axes sont perpendiculaires les uns aux autres et de longueur 1)

3.1.2 Pourquoi main droite ?

Regardez votre main droite.

- Votre pouce représente l'axe X,
- votre index représente l'axe Y,
- votre majeur représente l'axe Z.

La direction dans laquelle pointe chacun de vos doigts définit le sens de chaque axe.

Image utilisateur cf(fr.openclassrooms.com/informatique/cours/decouvrez-ogre-3d/le-systeme-de-coordonnees-1)

3.1.3 Repère local, repère absolu

Pour l'instant, nous n'avons toujours pas défini comment le repère était placé dans la scène. C'est une question de convention adoptée pour les applications 3D pour le repère de la scène:

- l'axe Y dirigé vers le haut
- les axes X et Z dans un plan horizontal.

Seulement¹, la scène n'est pas la seule à avoir son repère. En effet, chaque objet possède son propre repère appelé repère local. Lorsque l'objet se déplace ou tourne sur lui-même, le repère local fait de même. L'orientation du repère local est la suivante:

¹comment faire pour sauter une ligne avant le "Seulement"?

- l'axe Y est dirigé vers le haut de l'objet, pour la scène,
- l'axe X est dirigé vers sa droite
- l'axe Z vers l'arrière de l'objet.

Ci-dessous², j'ai représenté en noir le repère de la scène, et en bleu le repère local de la voiture, en respectant la convention que j'ai donnée. Image utilisateur cf(fr.openclassrooms.com/informatique/cours/dcouvrez-ogre-3d/le-systeme-de-coordonnees-1)

Nous verrons à quoi servent ces différents repères lorsque l'on commencera à déplacer nos objets.

3.1.4 Yaw, pitch, roll

yaw pitch roll sont les désignations anglaises pour les rotations autour des axes Y, X et Z respectivement. On peut traduire ces termes par lacet (yaw), tangage (pitch) et roulis (roll), qui sont utilisés par exemple en aéronautique ou en navigation.

Ces trois termes se retrouveront dans les noms qu'Ogre donne aux méthodes permettant d'effectuer des rotations.

Voici tout de suite un schéma illustrant les rotations qui s'appliquent à chaque axe:

Image utilisateur cf(fr.openclassrooms.com/informatique/cours/dcouvrez-ogre-3d/le-systeme-de-coordonnees-1)

Tout comme l'orientation des axes de notre repère, il y a un sens direct et un sens indirect pour les rotations! Vous tournez dans le sens direct lors d'une rotation contraire³ au sens des aiguilles d'une montre.

Pour effectuer une rotation, on appelle la méthode correspondante pour le noeud:

```
node->yaw(Radian(Math::PI));
```

Ceci fera faire un demi-tour au noeud par rapport à son axe vertical tandis que les méthodes pitch() et roll() s'utilisent de façon analogue pour les autres axes.

La rotation se fait par défaut par rapport au repère local. Il faut renseigner le second paramètre si vous voulez qu'il en soit autrement (voir la section suivante).

Les angles doivent être entrés en radians. Pour utiliser tout de même des degrés dans Ogre, vous devrez utiliser la classe Degree. La ligne de code précédente est équivalente à ceci:

```
node->yaw(Degree(180));
```

3.2 Déplacer des objets

3.2.1 Bouger un noeud de scène

Nous allons maintenant déplacer notre tête d'ogre pour vérifier la théorie et enfin sortir notre tête de terre!

²comment faire pour sauter une ligne avant le "Ci-dessous"?

³"rotation contraire" j-à "sens direct", est ce correct?

Pour cela, nous allons donc passer par le noeud auquel est rattaché notre mesh. Celui-ci possède deux méthodes qui peuvent nous servir.

3.2.2 setPosition()

La méthode `setPosition()` prend en paramètres les trois coordonnées X, Y et Z du point auquel on désire placer le noeud. On peut aussi lui passer un `Vector3`, qui contiendra lui-même ces coordonnées.

Les deux codes suivants sont donc équivalents.

```
Vector3 position = Vector3(30.0, 50.0, 0.0);
node->setPosition(position);
```

ou

```
node->setPosition(30.0, 50.0, 0.0);
```

La tête s'est maintenant déplacée vers la droite de 30 unités et de 50 unités vers le haut.

3.2.3 translate()

La méthode `translate()` déplace le noeud par rapport à sa position actuelle plutôt que par rapport à l'origine de la scène.

Elle prend les mêmes paramètres que la méthode `setPosition()`, mais avec un paramètre supplémentaire, défini par défaut, indiquant le noeud par rapport auquel on va se déplacer.

```
node->translate(-30.0, 50.0, 0.0);
```

En ajoutant cette ligne après la précédente, notre objet se retrouve donc maintenant à la position (0, 100, 0) dans la scène, ce qui est suffisant pour qu'il surplombe son petit jardin.

Image utilisateur

Le paramètre supplémentaire (par rapport à `setPosition`) permet de définir par rapport à quel repère on va déplacer le noeud.

Les trois valeurs possibles sont:

```
Node::TS.LOCAL //va déplacer le noeud par rapport ←
                au repere local
Node::TS.PARENT //va déplacer le noeud au repere du←
                  noeud parent
Node::TS.WORLD //va déplacer le noeud au repere de ←
                 la scene, qui est le repere absolu.
```

⁴Pour présenter les 3 valeurs possibles ne devrais je pas faire une liste itemize plutôt que d'utiliser un bloc de code?

3.2.4 Concrètement, ça veut dire quoi ?

Tout à l'heure, lorsque l'on a effectué une translation, on l'a fait par défaut par rapport au repère TS_WORLD, c'est-à-dire avec les axes tels que je vous les ai présentés précédemment. Maintenant, nous pouvons déplacer notre noeud par rapport au repère local de son noeud père par exemple, ou bien même par rapport à son propre repère local.

Mais à quoi ça sert de s'embêter avec ces paramètres ? On risque de faire des erreurs si l'on se place par rapport à un repère différent de la scène!

Prenons un exemple. Vous avez un vaisseau spatial qui peut se trouver dans n'importe quelles position et orientation de l'espace. Comment savoir facilement dans quelle direction je dois faire ma translation pour qu'on le voit aller en avant ?

Réponse: je n'ai pas à m'en occuper! En effet, l'axe qui va de l'avant vers l'arrière du vaisseau est l'axe Z, dans son repère local. Par conséquent, je n'ai qu'à dire à mon vaisseau d'avancer le long de l'axe Z (dans le sens négatif pour aller à l'avant) par rapport à son repère local. Et Ogre s'occupera gentiment de faire les calculs pour placer mon vaisseau correctement dans la scène.

Image utilisateur <http://fr.openclassrooms.com/informatique/cours/dcouvrez-ogre-3d/deplacer-des-objets>

Si vous avez compris cela, le paramètre TS_PARENT devrait suivre tout seul. Reprenons notre engin spatial. Sur ce vaisseau, on trouve R2D2 en train de se déplacer vers la droite, correspondant donc à l'axe X local du noeud du vaisseau. Pour effectuer cette translation, je n'ai qu'à demander à Ogre de déplacer mon robot le long de l'axe des abscisses par rapport au noeud du vaisseau (qui serait logiquement le noeud parent).

Vous commencez à comprendre l'intérêt des relations de parenté entre les noeuds ?

3.3 La caméra

La caméra, c'est l'élément qui définit la position de notre point de vue dans la scène, dans quelle direction on regarde, mais aussi jusqu'à quelle distance il est possible de voir s'afficher les objets éloignés.

Comme tous les éléments de base, un attribut caméra est présent dans la classe ExampleApplication. Sans elle nous n'aurions pas encore pu voir notre scène, vu que nous n'avons rien fait pour la créer!

3.3.1 Crédation

La caméra est créée par la méthode `createCamera()` de la classe ExampleApplication, nous allons tout de suite redéfinir cette méthode pour partir sur des bases connues. L'attribut correspondant à la caméra est appelé `mCamera`, nous pouvons donc l'utiliser pour créer notre caméra.

Comme c'est un objet qui se trouve dans la scène, nous allons passer par le `sceneManager`. Comme pour les noeuds ou les entités, vous pourrez donner un nom à votre caméra sous forme d'une chaîne de caractères.

Ajoutez la méthode `createCamera()` à votre classe `PremiereApplication` et ajoutez-y la ligne suivante.

```
mCamera = mSceneMgr->createCamera( "Ma Camera" );
```

3.3.2 Placement

Maintenant, il va nous falloir placer la caméra et l'orienter. Le placement se fait avec la méthode `setPosition()`.

La seconde méthode utilisée s'appelle `lookAt()` et, comme son nom l'indique, elle permet de déterminer le point de la scène que regarde notre caméra. On lui fournit un `Vector3` ou bien trois réels correspondant aux coordonnées désirées.

```
//placement de la camera
mCamera->setPosition( Vector3(-100.0, 150.0, 200.0));
//point de la scene que regarde notre camera
mCamera->lookAt( Vector3(0.0, 100.0, 0.0));
```

Enfin, on peut aussi indiquer les distances `near clip` et `far clip`, qui sont les distances minimale et maximale⁵ auxquelles doit se trouver un objet pour être affiché à l'écran.

```
mCamera->setNearClipDistance(1);
mCamera->setFarClipDistance(1000);
```

3.3.3 Code

PremiereApplication.cpp

Listing 3.1: PremiereApplication.cpp: Création de la caméra

```
#include "PremiereApplication.h"

void PremiereApplication::createScene()
{
    //creation d une entite
    Entity *head= mSceneMgr->createEntity("Tete", "ogrehead.mesh");

    //creation d un noeud
    SceneNode *node= mSceneMgr->getRootSceneNode()-><->
        createChildSceneNode("nodeTete", Vector3::ZERO, Quaternion::IDENTITY);

    node->yaw(Radian(Math::PI));
    node->yaw(Radian(Math::PI));
```

⁵"les distances minimale et maximale" il faut pas de "s" à "max/minimale""

```

//setPosition place le noeud aux coord passées en ↵
paramètres
Vector3 position = Vector3(30.0, 50.0, 0.0);
node->setPosition(position);

node->setPosition(30.0, 50.0, 0.0);
/* équivalent à
Vector3 position = Vector3(30.0, 50.0, 0.0);
node->setPosition(position);
*/

//déplace le noeud par rapport à sa position ↵
actuelle
node->translate(-30.0, 50.0, 0.0); //par défaut la ↵
translt se fait par rap à TS_WORLD

//attachement de l entité au noeud
node->attachObject( head );

//création d un plan
Plane plan(Vector3::UNIT_Y, 0);

//création d un mesh cad l objet 3d visible ds la ↵
scene
MeshManager::getSingleton().createPlane("sol",
    ResourceGroupManager::←
        DEFAULT_RESOURCE_GROUP_NAME,
    plan, 500, 500, 1, 1, true, 1, 1, 1, ←
        Vector3::UNIT_Z);

//entité qui représentera le plan
Entity *ent= mSceneMgr->createEntity("EntiteSol", "←
sol");

//ajout du matériau à l entité
ent->setMaterialName("Examples/GrassFloor"); //←
texture de pelouse
/* les différents matériaux sont sous /media/←
materials/scripts , par ex:
ent->setMaterialName("Examples/WaterStream"); //←
texture d eau animée*/

//création d un noeud
node = mSceneMgr->getRootSceneNode()->←
    createChildSceneNode();
node->attachObject(ent);
}

/* définit la position de notre point de vue*/
void PremiereApplication::createCamera()

```

```
{
    //creation de la camera
    mCamera = mSceneMgr->createCamera("Ma Camera");

    //position de la camera
    mCamera->setPosition(Vector3(-100.0, 150.0, 200.0))←
    ;

    //permet de determiner le point de la scene que ←
    //regarde notre camera
    mCamera->lookAt(Vector3(0.0, 100.0, 0.0));

    //definition des distances de near clip et de far ←
    //clip, qui
    //sont les distances minimale et maximale ←
    //auxquelles doit se
    //trouver un objet pour etre afficher a l'ecran.
    mCamera->setNearClipDistance(1);
    mCamera->setFarClipDistance(1000);
}
```

PremiereApplication.h

Listing 3.2: PremiereApplication.h: Création de la caméra

```
using namespace std;

#include <ExampleApplication.h>

class PremiereApplication : public ExampleApplication
{
public:
    void createScene();
    void createCamera();
};
```

main.cpp

Listing 3.3: main.cpp: Création de la caméra

```
#include <Ogre.h>

#include "PremiereApplication.h"

#if OGREPLATFORM == PLATFORM_WIN32 || OGREPLATFORM ==←
    OGRE_PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
#include "windows.h"
```

```

INT WINAPI WinMain(HINSTANCE hInst , HINSTANCE, LPSTR ↪
                     strCmdLine , INT)
#else
int main(int argc , char **argv)
#endif
{
    PremiereApplication app;

    try {
        app.go();
    } catch(Ogre::Exception& e) {
#if OGREPLATFORM == OGRE_PLATFORM_WIN32
        MessageBoxA(NULL, e.getFullDescription().c_str() ↪
                   () , "An exception has occurred!" , MB_OK | ↪
                   MB_ICONERROR | MB_TASKMODAL);
#else
        fprintf(stderr, "An exception has occurred: %s\n" ,
                e.getFullDescription().c_str());
#endif
    }

    return 0;
}

```

3.4 Le viewport

Une zone de rendu est une portion de l'écran sur laquelle est affichée ce que voit la caméra. La gestion de l'affichage dans une zone de rendu est à la charge de la classe Viewport.

Cette gestion de la zone de rendu a cela d'important que:

- la façon dont la caméra rend à l'écran ce qu'elle voit ne dépend pas que d'elle. En effet, la taille de votre zone de rendu et son format seront répercutés sur la portion de scène qu'il vous sera donné de voir. Si l'on ne tenait pas compte de ces paramètres, on pourrait obtenir une image aplatie si l'on élargissait la zone de rendu, ou bien au contraire compressée si l'on diminuait la largeur en laissant la hauteur constante.
- sur une même écran, vous pouvez afficher le rendu de plusieurs caméras dans la scène, voire des caméras de différents sceneManager.

Nous allons redéfinir la méthode `createViewports()` qui s'occupait jusqu'alors de ce travail pour nous. On commence par ajouter une vue:

```
Viewport *vue = mWindow->addViewport(mCamera);
```

Ici, `mWindow` est la fenêtre de notre application Ogre, c'est une instance de la classe `RenderWindow` dont nous verrons les détails dans un prochain chapitre.

La méthode addViewport permet donc d'ajouter une vue, seul son premier argument est obligatoire. Ce premier argument est la caméra à partir de laquelle est rendu la scène affichée dans la vue.

Nous allons faire coïncider le rapport largeur/hauteur de notre caméra avec celui du Viewport, pour avoir une image non déformée:

```
mCamera->setAspectRatio( Real(vue->getActualWidth()) / ←
    Real(vue->getActualHeight()) );
```

On applique un cast vers le format Ogre::Real pour obtenir un ratio décimal. Dans le cas contraire, le ratio serait tronqué pour être entier et la tête de notre ogre favori serait déformée.

Sachez aussi que c'est le Viewport qui définit la couleur de fond de la scène que vous voyez.

```
vue->setBackgroundColour( ColourValue( 0.0 , 0.0 , 1.0 ) );
```

Voici donc ma méthode createViewports() au complet:

Listing 3.4: Création d'un viewport

```
void PremiereApplication :: createViewports ()
{
    Viewport *vue = mWindow->addViewport (mCamera);

    //nous faisons coincider le rapport largeur / ↔
     hauteur de
    //notre camera avec celui du Viewport, pour avoir
    //une image non deformee
    mCamera->setAspectRatio( Real(vue->getActualWidth()) ←
        / Real(vue->getActualHeight()) );

    //couleur de fond de la vue
    vue->setBackgroundColour( ColourValue( 0.0 , 0.0 , 1.0 ) ←
        );
}
```

Maintenant, notre scène possède un magnifique ciel fond bleu.

3.4.1 Plusieurs viewports

La méthode addViewport permet d'ajouter des vues à notre scène, on peut donc avoir plusieurs vues à l'écran. Pour cela il faut renseigner les autres paramètres de la méthode addViewport.

Voici le prototype de la méthode addViewport.

Listing 3.5: Prototype de addViewport

```
virtual Viewport* Ogre::RenderTarget :: addViewport (
    Camera *           cam ,
    int      ZOrder = 0 ,
    float   left  = 0.0f ,
```

```
    float top = 0.0f,
    float width = 1.0f,
    float height = 1.0f
)
```

Les paramètres sont les suivants⁶:

- ZOrder: ordre relatif des vues les unes par rapport aux autres, les Z-orders les plus élevés sont au-dessus des autres. Le nombre donnée est pas important en lui-même car il s'agit d'une relation avec les autres Z-order, ainsi peut-on utiliser des nombres qui ne suivent pas.
- left: la position relative de la gauche du viewport sur la cible⁷ (entre 0 et 1).
- top: la position relative du haut du viewport sur la cible (entre 0 et 1).
- width: la largeur relative du viewport sur la cible (entre 0 et 1).
- height: la hauteur relative du viewport sur la cible (entre 0 et 1).

Code: 2 vues

Le code suivant permet la création de deux viewports:

Listing 3.6: createViewports: création de plusieurs vues

```
void PremiereApplication::createViewports()
{
    //la creation du Viewport "principal"
    Viewport *vue = mWindow->addViewport(mCamera, 0, 0,
                                           0, 0.8, 0.8);
    mCamera->setAspectRatio(Real(vue->getActualWidth()) /
        Real(vue->getActualHeight()));
    vue->setBackgroundColour(ColourValue(0.980, 0.502,
                                         0.447)); //saumon

    // creation d'un second viewport
    Viewport* vue2 = mWindow->addViewport(mCamera, 1,
                                           0.5, 0, 0.2, 0.2);
    vue2->setBackgroundColour(ColourValue(0.561, 0.737,
                                         0.561)); //darkseagreen
}
```

Et j'obtiens:

⁶d'autres essais et recherches seraient nécessaires pour la pleine compréhension de ces paramètres

⁷"cible" est visiblement le viewport principal

**Code: 3 vues**

Le code suivant permet la création de deux viewports:

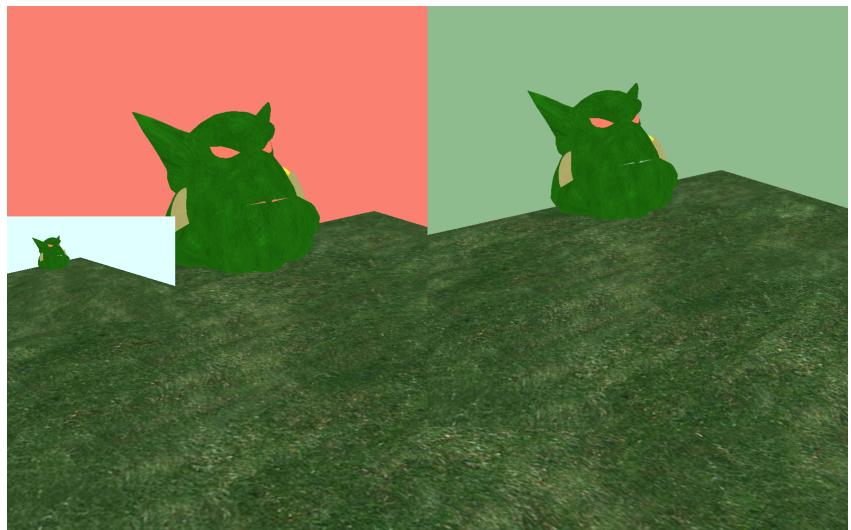
Listing 3.7: createViewports: création de plusieurs vues

```
void PremiereApplication::createViewports()
{
    //Viewport "principal"
    Viewport *vue = mWindow->addViewport(mCamera);
    mCamera->setAspectRatio(Real(vue->getActualWidth()) -->
        / Real(vue->getActualHeight()));
    vue->setBackgroundColour(ColourValue(0.980, 0.502, -->
        0.447)); //saumon

    //seconde vue
    Viewport* vue2 = mWindow->addViewport(mCamera, 1, -->
        0.5, 0, 0.8, 0.8);
    vue2->setBackgroundColour(ColourValue(0.561, 0.737, -->
        0.561)); //darkseagreen

    //troisieme vue
    Viewport* vue3 = mWindow->addViewport(mCamera, -->
        4, 0, 0.4, 0.2, 0.2);
    vue3->setBackgroundColour(ColourValue(0.878, 1.000, -->
        1.000)); //
```

Et j'obtiens:



Chapter 4

La lumières

4.1 Les lumières

4.1.1 Quelques fonctions de base

La lumière est nécessaire pour pouvoir voir quelque chose dans une scène. Comment a-t-on donc pu voir nos objets depuis le début de ce cours?

Il existe une propriété du scène Manager qui permet de définir une lumière ambiante. Cela permet d'éclairer la scène de façon homogène avec une certaine luminosité. Par défaut, on a un éclairage à la lumière blanche qui permet de voir ce qui se passe dans la scène. La ligne suivante peut être ajoutée au début de la méthode createscene() pour appliquer une lumière ambiante noire nous pourrons ainsi définir des lumières et voir leur influence.

```
msceneMgr->setAmbientLight( ColourValue( 0.0 , 0.0 , 0.0 ) );
```

La classe ColourValue permet de définir une couleur en entrant les quantités respectives de rouge, de vert puis de bleu dans un nombre compris entre 0 et 1. Il est aussi possible de définir une composante alpha (la transparence), utile pour des textures par exemple.

Une lumière est un objet de scène, on passe donc par le scène manager pour créer une lumière:

```
Light *light = msceneMgr->createLight( "lumiere1" );
```

Par défaut, la lumière créée est de type ponctuelle. Vous avez plus de détails sur les différents types de lumière un peu plus bas.

Mettons tout de suite en place quelques paramètres de base: la couleur émise (diffuse et spéculaire, que nous détaillerons dans le chapitre sur les matériaux) et la position.

```
light->setDiffuseColour( 1.0 , 0.7 , 1.0 );
light->setSpecularColour( 1.0 , 0.7 , 1.0 );
light->setPosition( -100 , 200 , 100 );
```

Les deux premiers paramètres sont les couleurs diffuses et spéculaires, au format RVB, avec des valeurs qui doivent être comprises entre 0 et 1.

La couleur diffuse est la couleur sous laquelle vont apparaître les objets non brillants, et la couleur spéculaire est un paramètre supplémentaire pour les matériaux réfléchissants comme le métal ou le verre. Pour une lumière, on met généralement la même couleur pour ces deux paramètres.

Vient ensuite la méthode `setPosition()`, qui ne devrait pas vous poser de problèmes, et enfin une dernière ligne¹ permettant d'amplifier ou de diminuer l'intensité lumineuse. Par défaut, ce coefficient est de 1, mais pour notre scène j'ai voulu l'augmenter pour qu'on y voie un peu plus clair: n'hésitez pas à jouer un peu avec pour faire des essais.

Enfin, sachez qu'il est possible d'attacher une lumière à un noeud de scène. Dans ce cas, la méthode `Light::setPosition()` définit la position relative de la lumière par rapport au noeud.

```
node->attachObject( light );
```

Vous pouvez donc facilement placer une lumière à la position d'un mesh censé émettre de la lumière - par exemple les phares d'une voiture - et les déplacer en même temps grâce à une seule commande vers le noeud de scène!

4.1.2 Les types de lumières

Ogre peut gérer différents types de lumières selon l'effet désiré. Ils sont au nombre de 3:

- la lumière ponctuelle: cette lumière émet dans toutes les directions à partir de sa position;
- la lumière directionnelle: une lumière dont les rayons vont dans une direction unique et qui n'a pas de position. C'est le genre de lumière qui permet de reproduire l'éclairage du soleil par exemple;
- le projecteur ou spot: c'est une lumière qui émet un cône lumineux à partir de sa position, à la façon d'une lampe-torche.

4.1.3 Lumière ponctuelle

C'est le type de lumière créé par défaut que l'on a vu plus haut. Pour le modifier manuellement, il faut utiliser le type `LT_POINT`.

```
light->setType( Light ::LT_POINT );
```

Avec ce type de lumière, il existe une méthode nous permettant aussi de limiter la portée de notre éclairage. Voici le prototype:

```
Light ::setAttenuation( Real range , Real constant , Real ←
    linear , Real quadratic )
```

C'est plus délicat car les paramètres doivent être choisis avec soin.

- Le premier est la distance caractéristique d'atténuation, c'est-à-dire la distance à partir de laquelle la luminosité diminue.

¹quelle dernière ligne ???

- constant est une constante d'atténuation comprise entre 0 et 1. Plus elle est proche de 0, et plus le passage de la lumière à l'ombre est brutal.
- linear et quadratic sont les paramètres de la courbe d'atténuation, et doivent être assez faibles, sinon la lumière s'atténue trop rapidement.

Ça ne marche pas! L'ogre est bien éclairé mais le sol reste désespérément noir!

L'atténuation ajoute une caractéristique un peu différente pour la gestion de la lumière. En effet, l'éclairage des surfaces est calculé en fonction des vertices situés dans la zone d'éclairage. Lorsqu'un vertice est dans la zone d'éclairage du spot, la surface autour de lui est éclairée, sinon elle est dans l'ombre. Ogre se charge ensuite de faire les dégradés entre les vertices plus ou moins éclairés.

Mais ici, notre plan n'est constitué que de quatre vertices (les coins), dont aucun n'est éclairé par le spot. Le sol n'est donc pas éclairé.

Pour régler ça, il suffit de modifier notre sol pour qu'il possède plus de vertices. Retrouvez la définition du plan et modifiez les paramètres de découpage pour obtenir 10 segments en largeur et en longueur (les deux paramètres avant le true).

```
Plane plan(Vector3::UNIT_Y, 0);
MeshManager::getSingleton().createPlane("sol", ←
    ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, ←
    plan, 500, 500, 10, 10, true, 1, 1, 1, Vector3::←
    UNIT_Z);
```

Notez que plus il y aura de vertices sur le modèle, plus ce sera précis, mais ce sera un peu plus coûteux en ressources.

4.1.4 Lumière directionnelle

Étant donné que cette lumière est de type "soleil" et qu'elle émet à l'infini, il n'est pas utile de renseigner sa position. En revanche, la méthode setDirection() permet de définir le vecteur directeur des rayons lumineux.

```
light->setType(Light::LT_DIRECTIONAL);
light->setDirection(10.0, -20.0, -5);
```

4.1.5 Projecteur

Le projecteur permet généralement de simuler un éclairage artificiel en proposant une lumière directionnelle définie dans un cône central et un cône extérieur, avec deux intensités différentes. Le cône central définit une lumière plus forte que le cône extérieur, où la lumière est quelque peu atténuée. Ces deux cônes sont définis par leur angle d'ouverture, ainsi que par un falloff, c'est-à-dire un coefficient indiquant si la transition entre les deux cônes doit être plus ou moins rapide:

```
light->setType(Light::LT_SPOTLIGHT);
light->setPosition(0, 150, -100);
light->setDirection(0, -1, 1);
light->setSpotlightRange(Degree(30), Degree(60), 1.0);
```

Notez que l'éclairage du spot obéit aux mêmes règles que pour l'atténuation d'une lumière ponctuelle: il faut que les vertices soient éclairés pour que l'éclairage soit visible.

De même que pour les lumières ponctuelles, vous pouvez ajouter une portée limitée à votre projecteur avec la méthode `setAttenuation()`.

4.2 Les ombres

4.2.1 Activer les ombres

Tout d'abord, on doit paramétriser nos lumières et nos entités pour projeter (ou non) des ombres. Que ce soit pour les lumières ou les entités, on utilise la même méthode pour choisir d'activer ou non la projection:

```
light->setCastShadows(true);
head->setCastShadows(true);
```

Si vous voulez qu'une entité ne projette aucune ombre, il suffit de mettre le paramètre à false. De même si vous voulez qu'une lumière ne projette aucune ombre pour les entités (pour une lumière d'ambiance ou d'ajustement, par exemple).

N'oubliez pas de désactiver la projection d'ombres pour le sol. D'une part parce que celui-ci n'a pas besoin de projeter d'ombres, d'autre part parce que certaines techniques nécessitent d'avoir ce paramètre désactivé pour avoir une ombre sur le mesh.

Avant de pouvoir afficher les ombres, il faut les activer. Cela se fait dans le scène Manager, par exemple:

```
msceneMgr->setShadowTechnique(Ogre::SHADOWTYPE_STENCIL_ADDITIVE);
```

On définit ici la technique de rendu qui sera utilisée pour les ombres dans la scène (voir ci-dessous).

4.2.2 Les différents types d'ombres

Ogre permet de générer différents types d'ombres selon les besoins, qui dépendent généralement des modèles concernés par la projection d'ombres.

Il existe deux techniques pour la génération d'ombres:

- le type "Stencil" (pochoir en anglais)
- le type "Texture"

Les ombres de type Stencil sont très précises dans les contours et permettent une très bonne projection d'ombre lorsque l'on y regarde de près. En revanche, elles sont assez coûteuses en ressources, notamment lorsque les mesh sont animés. Enfin, elles ne prennent pas du tout en compte la transparence des textures, un cube de texture transparente projettera donc une ombre si ce paramètre est activé. Les ombres de type Texture permettent de gérer la transparence des textures et sont moins coûteuses en ressources, mais leur précision est plus faible.

Enfin, chacune de ces deux catégories est composée de deux techniques, l'une dite modulative, l'autre additive. On obtient ainsi quatre techniques possibles:

- SHADOWTYPE_TEXTURE_MODULATIVE
- SHADOWTYPE_TEXTURE_ADDITIVE
- SHADOWTYPE_STENCIL_MODULATIVE
- SHADOWTYPE_STENCIL_ADDITIVE

Notez que dans chacun des cas, la technique additive est la meilleure, notamment pour une approche de type Stencil. La différence pour les techniques de type Texture est minime; en revanche, la technique Stencil additive permet d'obtenir des ombres plus ou moins sombres en fonction de l'éclairage grâce à des passes successives, tandis que la méthode Stencil modulative ne fait que projeter le modèle au sol une seule fois pour chaque lumière.

C'est donc dans le scène Manager que l'on s'occupe de déterminer la technique de rendu des ombres. Par défaut, celles-ci ne sont pas rendues.

```
msceneMgr->setShadowTechnique(Ogre::SHADOWTYPE_STENCIL_MODULATIVE);
```

Il n'est possible d'avoir qu'une seule technique enregistrée à la fois dans un scène Manager. Il n'est pas possible de choisir les types d'ombres à générer pour chaque lumière de la scène. Il faut donc faire un choix pour l'ensemble de vos ombres.

Ci-dessous, l'approche basée sur la texture, peu précise mais peu coûteuse en ressources:

Image utilisateur <http://fr.openclassrooms.com/informatique/cours/decouvrez-ogre-3d/les-ombres-1>

4.2.3 Code

Dans le code ci-dessous a été ajoutée une méthode pour la création d'une lumière et les ombres. Les objets qui projettent des ombres doivent être activé, ici seule la tête de ogre et la lumière elle-même projettent des ombres.

PremiereApplication.h

Listing 4.1: PremiereApplication.h: ajout d'une méthode pour la gestion de lumière et des ombres

```
using namespace std;

#include <ExampleApplication.h>
#include <OgreMovableObject.h>

class PremiereApplication : public ExampleApplication
{
    public:
        void createScene();
```

```

void createCamera();
void createViewports();

void createLux( std :: string , MovableObject * );
};
```

PremiereApplication.cpp

Listing 4.2: PremiereApplication.cpp: ajout d'une méthode pour la gestion de lumière et des ombres

```

#include "PremiereApplication.h"

void PremiereApplication :: createScene()
{
    //creation d une entité
    Entity *head= mSceneMgr->createEntity("Tete", "←
        ogrehead.mesh" );

    //creation d un noeud
    SceneNode *node= mSceneMgr->getRootSceneNode( )->←
        createChildSceneNode( "nodeTete" , Vector3 ::←
            ZERO, Quaternion ::IDENTITY);

    node->yaw(Radian(Math :: PI));
    node->yaw(Radian(Math :: PI));

    //setPosition place le noeud aux coord passées en ←
       paramètres
    Vector3 position = Vector3(30.0, 50.0, 0.0);
    node->setPosition(position);

    node->setPosition(30.0, 50.0, 0.0);
    /* équivalent à
    Vector3 position = Vector3(30.0, 50.0, 0.0);
    node->setPosition(position);
    */

    //déplace le noeud par rapport à sa position ←
       actuelle
    node->translate(-30.0, 50.0, 0.0); //par défaut la ←
       translt se fait par rapport à TS_WORLD

    //attachement de l entité au noeud
    node->attachObject(head);

    //creation d un plan
    Plane plan(Vector3 ::UNIT_Y, 0);
```

```

//creation d un mesh cad l objet 3d visible ds la ←
scene
MeshManager::getSingleton().createPlane("sol", ←
ResourceGroupManager::←
DEFAULTRESOURCEGROUPNAME, plan, 500, 500, 10,←
10, true, 1, 1, 1, Vector3::UNIT_Z);

//entite qui representera le plan
Entity *ent= mSceneMgr->createEntity("EntiteSol", "←
sol");

//ajout du materiau a l entite
ent->setMaterialName("Examples/GrassFloor");//←
texture de pelouse
/*les differents materiaux sont sous /media/←
materials/scripts , par ex:
ent->setMaterialName("Examples/WaterStream");//←
texture d eau animee*/

//creation d un noeud
node = mSceneMgr->getRootSceneNode()->←
createChildSceneNode();
node->attachObject(ent);

createLux("ponctuelle", head);//lumiere ponctuelle
//createLux("directionnelle", head);//lumiere ←
//directionnelle
//createLux("spot", head);//lumiere projecteur
}

/*
cree une lumiere selon le parametre passe:
createLux("ponctuelle"); -> lumiere ponctuelle
createLux("directionnelle"); -> lumiere ←
directionnelle
createLux("spot"); -> lumiere projecteur

le second parametre est l entite pour laquelle on ←
active les ombres

une lumiere noire est cree au debut de la methode

une ombre est cree en fin de methode
*/
void PremiereApplication::createLux(std::string ←
prmLightType, MovableObject * prmEnt)
{
    //application d une couleur noire
    mSceneMgr->setAmbientLight(ColourValue(0.0, 0.0, ←
0.0));
}

```

```

//definition d une lumiere
Light *light= mSceneMgr->createLight("lumiere1");

if (prmLightType == "ponctuelle")
{
    //definition du type de lumiere
    light->setType(Light::LT_POINT); //lumiere ← ponctuelle

    //definition de la position de la lumiere
    light->setPosition(-100, 200, 100);
}
else if (prmLightType == "directionnelle")
{
    light->setType(Light::LT_DIRECTIONAL); //lumiere← directionnelle
    light->setDirection(10.0, -20.0, -5); //vecteur ← directeur de la lumiere directionnelle

    //definition de la position de la lumiere
    light->setPosition(-100, 200, 100);
}
else
{
    light->setType(Light::LT_SPOTLIGHT); //lumiere ← directionnelle
    light->setDirection(0.0, -1, 1); //vecteur ← directeur de la lumiere directionnelle
    light->setSpotlightRange(Degree(30), Degree(60)←
        , 1.0);
}

//definition des couleur des lumieres diffuse
light->setDiffuseColour(1.0, 0.7, 0.1);
//et speculaire
light->setSpecularColour(1.0, 0.7, 0.1);

//ombre
//activation de la projection des ombres
light->setCastShadows(true);
prmEnt->setCastShadows(true);

//activation des ombres
mSceneMgr->setShadowTechnique(Ogre::←
    SHADOWTYPE_STENCIL_ADDITIVE);

}

/* definit la position de notre point de vue*/

```

```

void PremiereApplication ::createCamera()
{
    //creation de la camera
    mCamera = mSceneMgr->createCamera("Ma Camera");

    //position de la camera
    mCamera->setPosition(Vector3(-100.0, 150.0, 200.0))←
    ;

    //permet de determiner le point de la scene que ←
    regarde notre camera
    mCamera->lookAt(Vector3(0.0, 100.0, 0.0));

    //definition des distances de near clip et de far ←
    clip, qui
    //sont les distances minimale et maximale ←
    auxquelles doit se
    //trouver un objet pour etre afficher a l ecran.
    mCamera->setNearClipDistance(1);
    mCamera->setFarClipDistance(1000);
}

void PremiereApplication ::createViewports()
{
    //la creation du Viewport, appelee par la fenetre ←
    et prenant en parametre la
    //camera concerne, le premier parametre est la ←
    camera de laquelle le contenu
    //du viewport sera rendu, ce paramatre est le seul ←
    obligatoire
    Viewport *vue = mWindow->addViewport(mCamera);
    //Viewport *vue = mWindow->addViewport(mCamera, 0, ←
    0, 0, 0.8, 0.8);

    //Grace a ce Viewport nouvellement cree, nous ←
    allons faire coincider
    //le rapport largeur / hauteur de notre camera avec←
    celui du
    //Viewport, pour avoir une image non deformee
    mCamera->setAspectRatio(Real(vue->getActualWidth())←
    / Real(vue->getActualHeight()));

    //on definit ici la couleur de fond
    //vue->setBackgroundColour(ColourValue(0.0, 0.0, ←
    1.0)); //bleu
    vue->setBackgroundColour(ColourValue(0.980, 0.502,←
    0.447)); //saumon

    // creation d'un viewport dans le coin bas gauche
}

```

```
//les parametres autres que le premier sont ↵
    obligatoires pour la definition
//de plusieurs viewport
Viewport* vue2 = mWindow->addViewport(mCamera, 1, 0, ↵
    0.8, 0.2, 0.2);
vue2->setBackgroundColour(ColourValue(0.561, 0.737, ↵
    0.561)); //darkseagreen
}
```

Chapter 5

La gestion des entrées

5.1 Les frame listeners

5.1.1 Des "écouteurs d'images" ?

Utilité

Lorsque vous gérez les entrées de l'utilisateur (et même pour faire des calculs divers durant l'exécution de votre programme), l'ordinateur effectue les instructions nécessaires entre deux images (ou frames en anglais), donc pendant un temps très court.

En pratique, le moteur fonctionne dans une boucle, qui ne fait qu'afficher une image, puis fait des calculs; et ainsi de suite, sans s'arrêter. Il est donc possible pour le programmeur de donner ses instructions avant qu'une image soit rendue, ou bien après, ou bien même pendant que la carte graphique fait le rendu graphique.

Lorsque l'on aura vu comment créer cette boucle de rendu, nous serons à même de donner les instructions de la manière dont nous le désirons. En attendant, je vais vous présenter une classe qui a l'avantage de permettre de faire tout ce que je viens de vous expliquer de façon très simple: le frame listener.

Les méthodes à connaître

Un frame listener est une classe interface qui possède trois méthodes, dont voici les prototypes:

- virtual bool frameStarted(const FrameEvent& evt): est appelée avant que la frame ne soit rendue (frameStarted)
- virtual bool frameRenderingQueued(const FrameEvent& evt): est appelée après que la frame ait été rendue (frameEnded)
- virtual bool frameEnded(const FrameEvent& evt): est appelée après que le processeur graphique ait reçu les instructions pour le rendu (frameRenderingQueued)

En créant un objet dérivé de la classe frame listener dans votre application et en réimplémentant ces méthodes virtuelles, vous avez donc la possibilité de demander à Ogre d'effectuer les calculs dont vous avez besoin à chaque image.

frameStarted() et frameEnded() sont très similaires, étant donné qu'elles ont pour seule différence d'être appelées respectivement au début et à la fin de la boucle de rendu. Mais comme on est dans une boucle, en réalité il ne se passe quasiment rien entre l'appel à frameEnded() et celui à frameStarted(). La différence peut être utile par exemple si vous avez un calcul qui semble plus logique d'effectuer après que l'image soit rendue plutôt qu'avant, mais ce n'est qu'une question de lecture du code selon moi.

En revanche la dernière (frameRenderingQueued) est plus subtile. Comme je l'ai dit, elle est appelée dès que la carte graphique reçoit les instructions nécessaires pour afficher l'image à rendre.

Vous le savez probablement, c'est une opération très coûteuse en ressources et c'est souvent ce qui ralentit les jeux vidéo mettant en jeu de nombreux effets graphiques. Pendant ce temps-là, le processeur central attend que ça se passe. Par conséquent, si vous appelez la méthode frameRenderingQueued() pour faire des calculs, vous évitez d'avoir un processeur peu occupé pendant que la carte graphique fait son boulot !

De manière générale, on pourra utiliser cette méthode pour des opérations lourdes dont on sait qu'elles seront répétées à chaque image, afin de rentabiliser l'utilisation du processeur.

Contrôle de l'exécution

La valeur de retour des méthodes d'un frame listener est un booléen récupéré par Ogre pour savoir s'il doit continuer (true) ou non (false) l'exécution du programme.

Utiliser plusieurs frame listeners

Il est possible de créer autant de frame listeners que vous le désirez, pour effectuer des opérations diverses. En revanche, il est conseillé de ne pas en abuser pour éviter de trop segmenter votre application, il peut être intéressant d'appeler d'autres fonctions à partir d'un frame listener plutôt que d'en créer trop.

Enfin, et c'est là le plus important: **L'ordre d'exécution des frame listeners est laissé aux soins du moteur. Vous n'avez AUCUN contrôle dessus !**

En d'autres termes, si vous avez besoin d'effectuer des opérations dans un ordre précis, ne les mettez pas dans des frame listeners différents, car vous ne pourrez pas décider de l'ordre d'exécution. Il faut alors laisser un seul frame listener gérer les opérations, ou ne pas passer par eux (ce sera possible lorsque nous attaquerons la boucle de rendu).

5.1.2 Le frame listener en pratique

Comme nous allons vouloir redéfinir les méthodes du frame listener, il faut en faire une classe dérivée. Vu que nous sommes dans la gestion des entrées, nous allons tout de suite préparer le terrain en créant une classe InputListener dérivant de ExampleFrameListener.

Je dérive ici de la classe ExampleFrameListener, elle-même dérivée de FrameListener, car la classe ExampleApplication met déjà en place une gestion des entrées et attend un ExampleFrameListener. Cette classe s'occupe aussi de construire les objets nécessaires à l'écoute des entrées souris/clavier, ce que nous aborderons ultérieurement. Cependant, la méthode de traitement reste identique, nous allons juste devoir redéfinir frameRenderingQueued() pour implémenter notre propre gestion des entrées à la place de celle prévue par ExampleFrameListener.

Listing 5.1: InputListener.h

```
#include 'ExampleFrameListener.h'
class InputListener: public ExampleFrameListener
{
public:
    InputListener(RenderWindow* win, Camera* cam, ←
                  sceneManager *sceneMgr, bool bufferedKeys = ←
                  false, bool bufferedMouse = false, bool ←
                  bufferedJoy = false);
    virtual bool frameRenderingQueued(const FrameEvent&←
                                       evt);

private:
    Ogre::sceneManager *msceneMgr; //pointeur sur le ←
                                    scene Manager, qui servira à retrouver des ←
                                    objets dans la scène.
    bool mToucheAppuyee;          //pour garder une ←
                                    trace de l'état dans lequel se trouve une touche←
                                    particulière.

    /*distance de déplacement de la camera et sa ←
       vitesse, puis des angles de rotation.*/
    Ogre::Real mMouvement;
    Ogre::Real mVitesse;
    Ogre::Real mVitesseRotation;

    /*angles de rotation.*/
    Ogre::Radian mRotationX;
    Ogre::Radian mRotationY;
};
```

Dans votre fichier InputListener.cpp, préparez le constructeur ainsi que l'implémentation des méthodes, avec un corps vide pour l'instant:

Listing 5.2: InputListener.cpp

```
/*
Arguments:
RenderWindow* win: votre RenderWindow pour l'←
                     application
Camera* cam: la camera que vous utilisez
sceneManager *sceneMgr: le scene Manager
```

```

bool bufferedKeys: indique si vous desirez utiliser le ←
    buffer pour le clavier
bool bufferedMouse: indique si vous desirez utiliser le←
    buffer pour la souris
bool bufferedJoy: indique si vous desirez utiliser le ←
    buffer pour le joystick
*/
InputListener :: InputListener(RenderWindow* win, Camera*←
    cam, sceneManager *sceneMgr, bool bufferedKeys, ←
    bool bufferedMouse, bool bufferedJoy)
    : ExampleFrameListener(win, cam, bufferedKeys, ←
        bufferedMouse, bufferedJoy)
{
    mSceneMgr = sceneMgr;
    mVitesse = 100;
    mVitesseRotation = 0.3;
    mToucheAppuyee = false;
}

```

Il n'y a pas de constructeur écrit pour les frame listeners, car c'est une classe interface¹², seules les trois méthodes que j'ai présentées plus haut importent. En revanche, la classe ExampleFrameListener possède un constructeur qui prépare le terrain pour l'utilisation des entrées, que j'appelle dans ma classe dérivée.

Les trois booléens en paramètres indiquent si vous désirez utiliser le buffer respectivement pour le clavier, la souris et le joystick. Comme ce sera l'objet de la dernière partie de ce chapitre, je mets par défaut false (dans le .h).

Dans le corps du constructeur, j'initialise mes attributs mSceneMgr, mVitesse et mToucheAppuyee, qui nous serviront par la suite.

Il n'y a pas d'attribut à ajouter dans notre classe PremiereApplication, la classe ExampleApplication contient déjà un pointeur sur un ExampleFrameListener.

5.2 OIS

Pour gérer les entrées de l'utilisateur, nous allons utiliser la bibliothèque OIS³, qui est distribuée par défaut avec le SDK d'Ogre. Comme je l'ai dit en introduction de ce cours, un moteur 3D n'a pas de méthodes pour gérer autre chose que ce qui s'affiche sur votre écran. C'est pourquoi nous utiliserons cette bibliothèque pour récupérer les actions du joueur.

Pour ce faire, OIS représente les périphériques d'entrée par des objets, qui sont les suivants:

- Mouse pour la souris;
- Keyboard pour le clavier;

¹que je sache il n'y a pas d'interface en C++

²pas de constructeur parce que c'est une classe interface ??

³Object Oriented Input System, c'est à dire: Système d'entrées orienté objet

- Joystick pour les joysticks ou manettes de jeu.

Qui dit nouvelle bibliothèque dit nouveau namespace ! Ces classes se trouvent donc dans l'espace de nom OIS.

Les touches du clavier et les boutons de la souris sont des énumérations, définies comme ceci:

- OIS::KC_NOMDELATOUCHE pour le clavier;
- OIS::MB_NOMDUBOUTON pour la souris.

Ce qui donne par exemple OIS::KC_A pour la touche 'A'⁴, ou OIS::MC_Left pour le clic gauche.

Une dernière chose bonne à savoir: les codes de touches d'OIS correspondent aux touches physiques d'un clavier QWERTY. Ce qui signifie par exemple que OIS::KC_A correspond à la touche Q sur votre clavier AZERTY.

Afin d'utiliser OIS, il faut inclure le header correspondant. Celui-ci se trouve dans le dossier OIS du dossier include, on ajoutera donc la ligne de préprocesseur suivante en tête du header de InputListener:

Listing 5.3: Include OIS

```
#include <OIS/OIS.h>
```

Bien sûr, vous pouvez aussi ajouter le répertoire OIS à la liste des includes de votre IDE pour éviter d'avoir à le préciser dans le code.

5.3 Allons-y sans buffer

5

5.3.1 Explications

Dans le constructeur de notre frame listener, je vous ai dit que l'on avait mis les paramètres concernant l'utilisation du buffer à false, car c'est l'objet de la dernière partie de ce chapitre. Mais que signifie le fait d'utiliser ou non le buffer ?

Lorsque vous appuyez sur une touche, le clavier envoie un signal à l'ordinateur pour lui dire qu'une touche est actuellement pressée, en précisant quelle touche est concernée. Ce signal est envoyé tant que la touche reste enfoncee.

De son coté, notre programme effectue sa boucle infinie, rendant les images et faisant les calculs demandés. Supposons que vous appuyez sur une touche à un instant donné. Lorsque l'ordinateur arrivera à l'instruction lui demandant

⁴la touche 'Entrée' s'appelle 'Return' en anglais, ne cherchez donc pas OIS::KC_ENTER, vous ne trouverez pas.

⁵Le tutorial du site de Ogre "Frame Listeners and Unbuffered Input" présente le point traité ici d'une autre manière Frame Listeners and Unbuffered Input

de regarder ce qui se passe sur le clavier, il va voir qu'une touche est enfoncée et cherchera à effectuer les opérations demandées, et ceci tant que la touche reste enfoncée. Mais il n'est pas possible pour l'ordinateur de faire seul la différence entre une touche enfoncée et une touche qui vient d'être enfoncée.

Nous allons donc voir comment régler ce problème "à la main", puis nous verrons l'utilisation du buffer, qui constitue une autre façon de traiter l'entrée.

5.3.2 Création du frame listener

Avant de passer à la suite, réimplémentez la méthode `createFrameListener()` présente dans `ExampleApplication` dans la classe `PremiereApplication` en ajoutant le prototype et la définition :

Listing 5.4: `PremiereApplication::createFrameListener`

```
void PremiereApplication :: createFrameListener()
{
    //creation du framelistener en utilisant le ctor ←
    //prepare plus tot
    mFrameListener= new InputListener(mWindow, mCamera,←
        mSceneMgr, false , false , false );
    //signale a l objet root que ns avons un nv frame ←
    //listener et qu il faudra l appeler
    mRoot->addFrameListener(mFrameListener);
}
```

Le root est l'élément de base de l'application Ogre qui s'occupe notamment de gérer les frame listeners, nous le reverrons plus tard en approfondissant le fonctionnement du moteur.

5.3.3 Déplacer la caméra

Tout d'abord, nous allons considérer le déplacement de caméra, pour lequel nous n'avons pas besoin de savoir si la touche vient d'être appuyée : c'est simplement son état actuel qui compte.

Premièrement, il nous faut récupérer l'état actuel du clavier et de la souris. Localisez la méthode `frameRenderingQueued()` de votre `InputListener` et insérez-y ceci :

```
if (mMouse)
    mMouse->capture();
if (mKeyboard)
    mKeyboard->capture();
```

Ces deux lignes permettent de mettre à jour nos objets pour obtenir le nom des touches enfoncées.

Vérifions d'abord si la touche Echap est utilisée, auquel cas nous quitterons l'application.

```
if (mKeyboard->isKeyDown( OIS :: KC_ESCAPE ))
    return false;
```

Nous devons ensuite mettre à jour la valeur de mMouvement, qui sera la distance parcourue par la caméra si une direction est choisie. Comme le nombre d'images par seconde est variable, nous utilisons la propriété timeSinceLastFrame de l'événement, multipliée par la vitesse de la caméra. Le produit de la vitesse par le temps écoulé nous donne donc la distance parcourue.

J'ai aussi créé un vecteur dans lequel nous allons enregistrer les déplacements à effectuer. En effet, on peut utiliser plusieurs touches en même temps, il faut donc additionner les directions demandées, et les conserver pour déplacer la caméra en une seule fois.

```
Ogre::Vector3 deplacement = Ogre::Vector3::ZERO;
mMouvement = mVitesse * evt.timeSinceLastFrame;
```

Nous allons utiliser les flèches du clavier et les touches Z, S, Q, D pour nous déplacer ; j'ai aussi implémenté les touches fléchées, qui sont une configuration alternative pour le déplacement. Il faut donc vérifier si les touches qui nous intéressent sont enfoncées :

```
// La touche A d'un clavier QWERTY correspond au Q sur ←
// un AZERTY
if (mKeyboard->isKeyDown(OIS::KC_LEFT) || mKeyboard->←
    isKeyDown(OIS::KC_A))
    deplacement.x -= mMouvement;

if (mKeyboard->isKeyDown(OIS::KC_RIGHT) || mKeyboard->←
    isKeyDown(OIS::KC_D))
    deplacement.x += mMouvement;

// W correspond au Z du AZERTY
if (mKeyboard->isKeyDown(OIS::KC_UP) || mKeyboard->←
    isKeyDown(OIS::KC_W))
    deplacement.z -= mMouvement;

if (mKeyboard->isKeyDown(OIS::KC_DOWN) || mKeyboard->←
    isKeyDown(OIS::KC_S))
    deplacement.z += mMouvement;
```

Attention aux signes ! Vous devez respecter ce que nous avons vu dans le chapitre sur les déplacements !⁶

Le déplacement de la caméra fonctionne, il ne manque plus que la rotation de celle-ci. Il nous suffit pour cela de récupérer le déplacement relatif depuis la dernière fois que la souris a bougé (depuis le dernier appel à frameRenderingQueued() donc).

Pour retrouver cette valeur, on passe successivement par les attributs suivants :

- le mouseState contenu dans l'objet souris, contenant diverses informations sur l'état de la souris ;
- l'axe que l'on désire observer : ici, ce sera X ou Y pour le yaw ou le pitch ;

⁶hein? de quoi parle t il?

- le déplacement relatif de la souris suivant cet axe.

Maintenant, occupons-nous du mouvement de la souris. Pour récupérer le déplacement de celle-ci, nous devons récupérer son état, comme indiqué dans le code suivant.

```
const OIS::MouseState &mouseState = mMouse->→
getMouseState();
```

à partir de cette référence on peut notamment récupérer le déplacement de la souris depuis la dernière image, en appelant l'axe X ou Y puis l'attribut rel.

```
mRotationX = Degree(-mouseState.Y.rel * ←
mVitesseRotation);
mRotationY = Degree(-mouseState.X.rel * ←
mVitesseRotation);
```

Il faut particulièrement faire attention aux axes et aux signes. Je considère que mRotationX (respectivement mRotationY) correspond à la rotation autour de l'axe X (respectivement Y), c'est-à-dire lorsque je déplace ma souris en avant ou en arrière (respectivement à gauche ou à droite). Or, le déplacement vers l'avant ou l'arrière de la souris correspond à son axe Y, c'est pour ça que je demande l'axe Y de la souris pour trouver la rotation autour de X dans l'espace 3D.

On rajoute une multiplication par la vitesse de rotation voulue et on convertit le tout en degrés, sinon le mouvement est bien trop rapide.

Enfin, on appelle les méthodes de rotation et de déplacement de la caméra :

```
mCamera->yaw(mRotationY);
mCamera->pitch(mRotationX);
mCamera->moveRelative(deplacement);
```

La dernière ligne, comme vous le remarquez, déplace la caméra par rapport à son repère local, ce qui évite de faire la transformation de la variable deplacement à la main. Il est aussi possible de demander un déplacement par rapport au repère absolu avec Camera::move().

5.3.4 Et avec un noeud de scène ?

J'en profite pour vous montrer comment on aurait procédé pour déplacer un noeud de scène par exemple, qui utilise la méthode translate().

Les deux lignes suivantes sont équivalentes :

```
node->translate(deplacement, TS_LOCAL);
node->translate(node->getOrientation() * deplacement, ←
TS_PARENT);
```

La première ligne est très similaire à celle utilisée pour la caméra, il suffit de préciser que l'on se déplace par rapport au repère local du noeud de scène.

La seconde solution indique un déplacement relatif au noeud parent, mais utilise le quaternion retourné par la méthode getOrientation() multiplié par le vecteur de déplacement pour obtenir la direction souhaitée dans ce repère. En pratique, on utilisera seulement la première ligne, plus courte et plus propre dans le code.

5.3.5

Mini-TP : créer un interrupteur

Pour gérer un événement qui ne doit arriver qu'une fois lorsque la touche est appuyée, il y a une précaution supplémentaire à prendre. Je vous ai dit plus haut que votre ordinateur ne retenait pas l'état dans lequel se trouvait votre clavier ou votre souris à l'image précédente. Cependant, rien ne nous empêche de le faire nous-mêmes !

à titre d'exemple, disons que l'on veut utiliser la touche T pour allumer et éteindre la lumière de notre scène. Il va donc falloir vérifier à chaque image si la touche T est enfoncée, et si en plus ce n'était pas déjà le cas à l'image précédente. Pour cela, on utilisera l'attribut mToucheAppuyee de notre classe InputListener.

Un indice: le SceneManager possède une méthode getLight() qui permet de récupérer un pointeur sur une lumière à partir du nom de celle-ci...

à vos claviers ! La réponse se trouve juste après.

Listing 5.5: Capture de l'état ponctuel d'une touche

```
bool etatTouche = mKeyboard->isKeyDown(OIS::KC_T);
if(etatTouche && !mToucheAppuyee)
{
    Ogre::Light *light = mSceneMgr->getLight("lumiere1");
    light->setVisible(!light->isVisible());
}
mToucheAppuyee = etatTouche;
```

Tout d'abord, je récupère l'état actuel de ma touche T dans une variable locale. Je vérifie si la touche est actuellement enfoncée et si elle ne l'était pas déjà à l'aide de l'attribut mToucheAppuyee. Si ma condition est vérifiée, je récupère ma lumière, et je change son état (visible ou non).

Enfin, j'enregistre l'état actuel de ma touche T dans mToucheAppuyee, en prévision de la prochaine image!

5.3.6 Code

Listing 5.6: InputListener.h

```
#include "ExampleFrameListener.h"

//nous utiliserons OIS pour gerer les entrees de l'utilisateur
#include <OIS/OIS.h>

//la classe ExampleFrameListener ExampleApplication met deja en oeuvre une gestion des entrees et attend un ExampleFrameListener
//ExampleFrameListener s'occupe aussi de construire les objets necessaires a l'ecoute des entrees
class InputListener : public ExampleFrameListener
```

```
{
public:
    InputListener(RenderWindow* win, Camera* cam, ←
                  SceneManager *sceneMgr,
                  bool bufferedKeys = false, bool ←
                  bufferedMouse = false,
                  bool bufferedJoy = false);

    //nous allons juste devoir redefinir ←
    //frameRenderingQueued() pour implementer ←
    //notre propre gestion des entrees a la place ←
    //de celle prevue par ExampleFrameListener.
    virtual bool frameRenderingQueued(const ←
                                       FrameEvent& evt);

private:
    Ogre::SceneManager *mSceneMgr;
    Ogre::Camera *mCamera;

    bool mContinuer;
    bool mToucheAppuyee;

    Ogre::Real mMouvement;
    Ogre::Real mVitesse;
    Ogre::Real mVitesseRotation;

    Ogre::Radian mRotationX;
    Ogre::Radian mRotationY;
};
```

Dans la méthode InputListener::frameRenderingQueued, les touches pressées et les mouvements de la souris permettent soit le déplacement de la tête soit le déplacement de la caméra selon les lignes qui en fin de méthodes sont commentées.

Listing 5.7: InputListener.cpp

```
#include <OIS/OIS.h>
#include "InputListener.h"

InputListener :: InputListener(RenderWindow* win, Camera*←
                               cam, SceneManager *sceneMgr,
                               bool bufferedKeys, bool ←
                               bufferedMouse, bool ←
                               bufferedJoy)
: ExampleFrameListener(win, cam, bufferedKeys, ←
                      bufferedMouse, bufferedJoy)

{
```

```

mCamera = cam;
mSceneMgr = sceneMgr;
mVitesse = 100;
mVitesseRotation = 0.3;
mToucheAppuyee = false;
}

bool InputListener::frameRenderingQueued(const ←
    FrameEvent& evt)
{
    //ces lignes permettent la mise a jour de nos ←
    //objets pour obtenir le nom des touches enfoncées
    if(mMouse){
        mMouse->capture();
    }
    if(mKeyboard){
        mKeyboard->capture();
    }

    if(mKeyboard->isKeyDown(OIS::KC_ESCAPE)){
        mContinuer = false;
    }
    else{
        mContinuer= true;
    }

    Ogre::Vector3 deplacement = Ogre::Vector3::ZERO;
    mMouvement = mVitesse * evt.timeSinceLastFrame;

    // La touche A d un clavier QWERTY correspond au Q ←
    // sur un AZERTY
    if ( mKeyboard->isKeyDown(OIS::KC_LEFT) || ←
        mKeyboard->isKeyDown(OIS::KC_A) ){
        deplacement .x -= mMouvement;
    }

    if ( mKeyboard->isKeyDown(OIS::KC_RIGHT) || ←
        mKeyboard->isKeyDown(OIS::KC_D) ){
        deplacement .x += mMouvement;
    }

    // W correspond au Z du AZERTY
    if ( mKeyboard->isKeyDown(OIS::KC_UP) || mKeyboard->isKeyDown(OIS::KC_W) ){
        deplacement .z -= mMouvement;
    }
}

```

```

    }

    if ( mKeyboard->isKeyDown(OIS::KCDOWN) || ←
        mKeyboard->isKeyDown(OIS::KC_S) ){
        deplacement.z += mMouvement;
    }

    /**
const OIS::MouseState &mouseState = mMouse->←
    getMouseState();
mRotationX = Degree(-mouseState.Y.rel * ←
    mVitesseRotation);
mRotationY = Degree(-mouseState.X.rel * ←
    mVitesseRotation);

//pour que la camera bouge selon les touches ←
    pressees et le mouvement de la souris
//mCamera->yaw(mRotationY);
//mCamera->pitch(mRotationX);
//mCamera->moveRelative(deplacement);

//pour que le noeud "nodeTete" bouge selon les ←
    touches pressees et le mouvement de la souris
mSceneMgr->getSceneNode("nodeTete")->yaw(mRotationY←
    );
mSceneMgr->getSceneNode("nodeTete")->pitch(←
    mRotationX);
mSceneMgr->getSceneNode("nodeTete")->translate(←
    deplacement, Ogre::Node::TS_LOCAL);

return mContinuer;
}

```

Listing 5.8: PremiereApplication.h

```

using namespace std;

#include <ExampleApplication.h>
#include <OgreMovableObject.h>

#include "InputListener.h"

class PremiereApplication : public ExampleApplication
{
    public:
        void createScene();
        void createCamera();
        void createViewports();

```

```

    void createFrameListener() ;
    void createLux( std::string , MovableObject * ) ;
};
```

Listing 5.9: PremiereApplication.cpp

```

#include "PremiereApplication.h"

void PremiereApplication::createFrameListener()
{
    mFrameListener = new InputListener(mWindow, mCamera
        , mSceneMgr, false, false, false);

    //root est l'element de base de l'application Ogre ←
    //qui s'occupe notamment de gerer les frames ←
    //listeners
    mRoot -> addFrameListener(mFrameListener);
}

void PremiereApplication::createScene()
{
    //creation d une entite
    Entity *head= mSceneMgr->createEntity("Tete", "←
        ogrehead.mesh" );

    //creation d un noeud
    SceneNode *node= mSceneMgr->getRootSceneNode()->←
        createChildSceneNode("nodeTete" , Vector3::ZERO, ←
        Quaternion::IDENTITY);

    node->yaw(Radian(Math::PI));
    node->yaw(Radian(Math::PI));

    //setPosition place le noeud aux coord passees en ←
    //parametres
    Vector3 position = Vector3(30.0, 50.0, 0.0);
    node->setPosition(position);

    node->setPosition(30.0, 50.0, 0.0);
    /*equivalent a
    Vector3 position = Vector3(30.0, 50.0, 0.0);
    node->setPosition(position);
    */

    //deplace le noeud par rapport a sa position ←
    //actuelle
```

```

node->translate(-30.0, 50.0, 0.0); //par defaut la ←
trnslt se fait par rap a TS_WORLD

//attachement de l entite au noeud
node->attachObject(head);

//creation d un plan
Plane plan(Vector3::UNIT_Y, 0);

//creation d un mesh cad l objet 3d visible ds la ←
scene
MeshManager::getSingleton().createPlane("sol",
    ResourceGroupManager::←
        DEFAULT_RESOURCE_GROUP_NAME,
    plan, 500, 500, 10, 10, true, 1, 1, 1, ←
    Vector3::UNIT_Z);

//entite qui representera le plan
Entity *ent= mSceneMgr->createEntity("EntiteSol", "←
sol");

//ajout du materiau a l entite
ent->setMaterialName("Examples/GrassFloor");//←
texture de pelouse
/*les differents materiaux sont sous /media/←
materials/scritps , par ex:
ent->setMaterialName("Examples/WaterStream");//←
texture d eau animee*/

//creation d un noeud
node = mSceneMgr->getRootSceneNode()->←
createChildSceneNode();
node->attachObject(ent);

createLux("ponctuelle", head); //lumiere ponctuelle
//createLux("directionnelle", head); //lumiere ←
directionnelle
//createLux("spot", head); //lumiere projecteur

}

/*
cree une lumiere selon le parametre passe:
createLux("ponctuelle"); -> lumiere ponctuelle
createLux("directionnelle"); -> lumiere ←
directionnelle
createLux("spot"); -> lumiere projecteur

une lumiere noire est cree au debut de la methode

```

```

une ombre est cree en fin de methode
*/
void PremiereApplication ::createLux( std::string ←
    prmLightType , MovableObject * prmEnt)
{
    //application d une couleur noire
    mSceneMgr->setAmbientLight(ColourValue(0.0, 0.0, ←
        0.0));

    //definition d une lumiere
    Light *light= mSceneMgr->createLight("lumiere1");

    if (prmLightType == "ponctuelle")
    {
        //definition du type de lumiere
        light->setType(Light ::LTPOINT); //lumiere ←
            ponctuelle

        //definition de la position de la lumiere
        light->setPosition(-100, 200, 100);
    }
    else if (prmLightType == "directionnelle")
    {
        light->setType(Light ::LTDIRECTIONAL); //lumiere←
            directionnelle
        light->setDirection(10.0, -20.0, -5); //vecteur ←
            directeur de la lumiere directionnelle

        //definition de la position de la lumiere
        light->setPosition(-100, 200, 100);
    }
    else
    {
        light->setType(Light ::LTSPOTLIGHT); //lumiere ←
            directionnelle
        light->setDirection(0.0, -1, 1); //vecteur ←
            directeur de la lumiere directionnelle
        light->setSpotlightRange(Degree(30), Degree(60)←
            , 1.0);
    }

    //definition des couleur des lumieres diffuse
    light->setDiffuseColour(1.0, 0.7, 0.1);
    //et speculaire
    light->setSpecularColour(1.0, 0.7, 0.1);

    //ombre
    //activation de la projection des ombres
    light->setCastShadows(true);
    prmEnt->setCastShadows(true);
}

```

```

//activation des ombres
mSceneMgr->setShadowTechnique( Ogre::←
    SHADOWTYPE_STENCIL_ADDITIVE );
}


void PremiereApplication ::createCamera()
{
    //creation de la camera
    mCamera = mSceneMgr->createCamera("Ma Camera");

    //position de la camera
    mCamera->setPosition( Vector3(-100.0, 150.0, 200.0))←
        ;
    //permet de determiner le point de la scene que ←
        regarde notre camera
    mCamera->lookAt( Vector3(0.0, 100.0, 0.0));

    //definition des distances de near clip et de far ←
        clip , qui
    //sont les distances minimale et maximale ←
        auxquelles doit se
    //trouver un objet pour etre afficher a l 'ecran.
    mCamera->setNearClipDistance(1);
    mCamera->setFarClipDistance(1000);
}

void PremiereApplication ::createViewports()
{
    //la creation du Viewport, appelee par la fenetre ←
        et prenant en parametre la
    //camera concernee, le premier parametre est la ←
        camera de laqll le contenu
    //du viewport sera rendu, ce paramatre est le seul ←
        obligatoire
    Viewport *vue = mWindow->addViewport(mCamera);
    //Viewport *vue = mWindow->addViewport(mCamera, 0, ←
        0, 0, 0.8, 0.8);

    //Grace a ce Viewport nouvellement cree , nous ←
        allons faire coincider
    //le rapport largeur / hauteur de notre camera avec←
        celui du
    //Viewport , pour avoir une image non deformee
    mCamera->setAspectRatio( Real(vue->getActualWidth())←
        / Real(vue->getActualHeight()));

    //on definit ici la couleur de fond
}

```

```

    vue->setBackgroundColour(ColourValue(0.0, 0.0, 1.0)←
        ); //bleu
    //vue->setBackgroundColour(ColourValue(0.980, ←
        0.502, 0.447)); //saumon

    // creation d'un viewport dans le coin bas gauche
    //les parametres autres que le premier sont ←
        obligatoires pour la definition
    //de plusieurs viewport
    Viewport* vue2 = mWindow->addViewport(mCamera, 1, 0,←
        0.8, 0.2, 0.2);
    vue2->setBackgroundColour(ColourValue(0.561, 0.737, ←
        0.561)); //darkseagreen
}

```

Puisque nous avons ajouté un fichier source il est nécessaire de modifier le fichier CMakeLists.txt tel que suit:

Listing 5.10: CMakeLists.txt

```

project(helloworld)
cmake_minimum_required(VERSION 2.6)

set(CMAKE_MODULE_PATH "/usr/share/OGRE/cmake/modules")

# Il faut indiquer à cmake où se trouvent les includes ←
# en question
#include_directories("/home/adkoba/Workspace/←
#                   Ogre_src_v1-8-1/Samples/Common/include")
include_directories("include")

# Bien sûr, pour compiler Ogre, il faut le chercher, et ←
# définir le répertoire contenant les includes.
find_package(OGRE REQUIRED)
include_directories(${OGRE_INCLUDE_DIRS})

# L'exemple dépend aussi de OIS, une lib pour gérer la ←
# souris, clavier, joystick...
find_package(OIS REQUIRED)

# On définit les sources qu'on veut compiler
SET(SOURCES
    src/InputListener.cpp
    src/PremiereApplication.cpp
    src/main.cpp
)
# On les compile
add_executable(
    premiereapp ${SOURCES}
)

```

```
)
# Et pour finir , on lie l'executable avec les librairies←
# que find_package nous a gentillement trouve.
target_link_libraries(premiereapp ${OGRELIBRARY} ${←
    OISLIBRARY} -lboost_system)

set( RESOURCESFILE
    media/
    plugins/
    resources/ogre.cfg
    resources/plugins.cfg
    resources/resources.cfg
)
# do the copying
foreach( file_i ${RESOURCESFILE})
    add_custom_command(
        TARGET premiereapp
        POST_BUILD
        COMMAND cp -R ${CMAKE_SOURCE_DIR}/${file_i} ${←
            CMAKE_BINARY_DIR}
        COMMENT "copy file ${file_i}"
    )
endforeach( file_i )
```

Comme nous pouvons le voir nous avons juste modifié la liste des sources du projet

5.4 Avec buffer, c'est plus simple ?

La méthode présentée ci-avant pour contrôler si une touche vient ou non d'être appuyée fonctionne mais est peu pratique si on doit surveiller quinze touches.

Heureusement, OIS a pensé à tout, nous allons donc voir une autre façon de faire ce que l'on vient juste d'écrire. On va commencer comme précédemment par le déplacement de la caméra, puis on verra comment gérer notre interrupteur.

5.4.1 Mise en place

Nous allons commencer par activer l'utilisation du buffer pour la souris et le clavier lors de la construction de notre frame listener. Il suffit pour cela de mettre les paramètres correspondants à true.

Listing 5.11: Activation du buffer pour la souris et le clavier

```
void PremiereApplication::createFrameListener()
{
    mFrameListener= new InputListener(mWindow, mCamera,←
        mSceneMgr, true, true, false);      mRoot->←
        addFrameListener( mFrameListener );
```

```
}
```

C'est quasiment tout , il va simplement falloir rajouter deux petites lignes dans le constructeur pour que tout soit prêt.

Afin d'utiliser le buffer, il faut fournir un objet (un "écouteur" dérivant d'une des classes OIS::***Listener selon le périphérique à écouter) qui sera celui qui recevra les événements du type "cette touche vient d'être appuyée, que dois-je faire ?". Pour cela, OIS fournit une méthode pour chacun de trois périphériques d'entrée (clavier, souris, joystick) :

- virtual void OIS::Mouse::setEventCallback(OIS::MouseListener* mouseListener);
- virtual void OIS::Keyboard::setEventCallback(OIS::KeyListener* keyListener);
- virtual void OIS::JoyStick::setEventCallback(OIS::JoyStickListener* joyStickListener);

Cette méthode prend donc en paramètre un pointeur sur un listener du périphérique que vous voulez utiliser. Nous allons donc rajouter deux classes mères à notre InputListener : OIS::MouseListener et OIS::KeyListener. Modifiez donc la déclaration de la classe InputListener :

Listing 5.12: Classes mères pour gestion des Listeners

```
class InputListener : public ExampleFrameListener, OIS←
    :: KeyListener, OIS :: MouseListener
```

Dans le constructeur, vous pouvez maintenant insérer les deux lignes suivantes (mMouse et mKeyboard sont déclarées dans ExampleFrameListener) :

Listing 5.13: Enregistrement des listener

```
mMouse->setEventCallback( this );
mKeyboard->setEventCallback( this );
```

Vous ne pouvez enregistrer qu'un seul écouteur par périphérique d'entrée. En cas d'appels multiples à la méthode setEventCallback(), c'est le dernier appel qui définit l'écouteur à utiliser. Pour que différents objets reçoivent les événements, il faudra donc les redistribuer à partir de l'écouteur receveur.

Au chapitre des modifications, supprimez les anciens attributs d'InputListener et mettez ceux-ci :

Listing 5.14: Attributs d'InputListener

```
private:
    Ogre :: SceneManager *mSceneMgr;
    bool mContinuer;
    Ogre :: Vector3 mMouvement;
    Ogre :: Real mVitesse;
    Ogre :: Real mVitesseRotation;
```

En initialisant ces attributs, votre constructeur devrait maintenant ressembler à ceci :

Listing 5.15: Constructeur d'InputListener

```

InputListener(RenderWindow* win, Camera* cam, ←
    SceneManager *sceneMgr, bool bufferedKeys = false, ←
bool bufferedMouse = false, bool bufferedJoy = false←
) : ExampleFrameListener(win, cam, bufferedKeys, ←
bufferedMouse, bufferedJoy)
{
    mSceneMgr = sceneMgr;
    mContinuer = true; //permettra d'enregistrer l'←
    appui sur la touche Echap
    mMouvement = Ogre::Vector3::ZERO; //vecteur de la ←
    direction ds laquelle se déplacer
    mVitesse = 100;
    mVitesseRotation = 0.2; //facteur multiplicatif pr ←
    ajuster la vitesse de la cam
    mMouse->setEventCallback(this);
    mKeyboard->setEventCallback(this);
}

```

L'attribut mContinuer permettra d'enregistrer l'appui sur la touche Echap, mMouvement sera le vecteur de la direction dans laquelle on doit se déplacer et mVitesseRotation un facteur multiplicatif permettant d'ajuster la vitesse de rotation de la caméra.

On met à jour la valeur de retour de la méthode frameRenderingQueued().

```

bool InputListener :: frameRenderingQueued( const Ogre::←
    FrameEvent& evt )
{
    if (mMouse)
        mMouse->capture();
    if (mKeyboard)
        mKeyboard->capture();

    return mContinuer;
}

```

Enfin, il y a des méthodes virtuelles pures à réimplémenter dans notre classe. Ces méthodes seront appelées lors de l'événement correspondant sur le clavier (touche enfoncée ou relâchée) ou sur la souris (bouton appuyé ou relâché, déplacement). De même que les méthodes des frame listeners d'Ogre, elles renvoient un booléen que l'on utilisera pour savoir si l'on doit interrompre le programme.

Ajoutons donc les prototypes dans notre header et un simple retour de valeur dans le corps des méthodes pour commencer.

Contrairement aux méthodes des frame listeners, la valeur de retour ne détermine pas si l'on doit continuer ou non l'exécution. C'est pour cela que l'on devra passer par l'attribut mContinuer pour surveiller l'appui sur la touche Echap.

Listing 5.16: Méthodes virtuelles appelées lors d'un évènement sur un périphérique

```

bool InputListener :: mouseMoved(const OIS :: MouseEvent &e)
{
    return true;
}

bool InputListener :: mousePressed(const OIS :: MouseEvent &e, OIS :: MouseButtonID id)
{
    return true;
}

bool InputListener :: mouseReleased(const OIS :: MouseEvent &e, OIS :: MouseButtonID id)
{
    return true;
}

bool InputListener :: keyPressed(const OIS :: KeyEvent &e)
{
    return true;
}

bool InputListener :: keyReleased(const OIS :: KeyEvent &e)
{
    return true;
}

```

On commence par implémenter la touche Echap. Si elle est appuyée, on passe simplement l'attribut mContinuer à false.

Listing 5.17: Implémentation de l'appui sur ECHAP

```

bool InputListener :: keyPressed(const OIS :: KeyEvent &e)
{
    switch(e .key)
    {
        case OIS :: KC_ESCAPE:
            mContinuer = false;
            break;
    }

    return mContinuer;
}

```

On gère ensuite l'appui sur les touches de déplacement en modifiant les composantes de mMouvement en fonction de la touche. On va aussi multiplier la vitesse de déplacement par deux lorsque l'on appuie sur la touche majuscule gauche.

Listing 5.18: Implémentation de l'appui sur les touches de déplacement

```
bool InputListener :: keyPressed (const OIS :: KeyEvent &e)
{
    switch(e.key)
    {
        case OIS :: KC_ESCAPE:
            mContinuer = false;
            break;
        case OIS :: KC_W:
            mMouvement.z -= 1;
            break;
        case OIS :: KC_S:
            mMouvement.z += 1;
            break;
        case OIS :: KC_A:
            mMouvement.x -= 1;
            break;
        case OIS :: KC_D:
            mMouvement.x += 1;
            break;
        case OIS :: KC_LSHIFT:
            mVitesse *= 2;
            break;
    }

    return mContinuer;
}
```

Enfin, dans la méthode keyReleased, on va "retirer" la composante que l'on ajoute lors de l'appui sur une touche. Le code est donc semblable, seuls les signes changent.

```
bool InputListener :: keyReleased (const OIS :: KeyEvent &e)
{
    switch(e.key)
    {
        case OIS :: KC_W:
            mMouvement.z += 1;
            break;
        case OIS :: KC_S:
            mMouvement.z -= 1;
            break;
        case OIS :: KC_A:
            mMouvement.x += 1;
            break;
        case OIS :: KC_D:
            mMouvement.x -= 1;
            break;
        case OIS :: KC_LSHIFT:
            mVitesse /= 2;
            break;
    }
```

```

    }

    return true;
}

```

Maintenant que l'on gère correctement l'évolution de nos variables de mouvement et de vitesse de déplacement, il faut écrire le déplacement de la caméra dans la méthode frameRenderingQueued().

Listing 5.19: Implémentation du déplacement de la caméra dans la méthode frameRenderingQueued

```

virtual bool frameRenderingQueued(const FrameEvent& evt←
)
{
    if(mMouse)
        mMouse->capture();

    if(mKeyboard)
        mKeyboard->capture();

    Ogre::Vector3 deplacement = Ogre::Vector3::ZERO;
    deplacement = mMouvement * mVitesse * evt.←
        timeSinceLastFrame;
    mCamera->moveRelative(deplacement);

    return mContinuer;
}

```

Pour la rotation de la caméra, tout se passe dans la méthode mouseMoved(), dont l'événement reçu en paramètre contient l'état de la souris, permettant comme précédemment de retrouver le déplacement relatif de la souris.

On multiplie cette valeur relative par la vitesse de rotation, on fait attention aux signes, et voici ce qu'on obtient :

Listing 5.20: Implémentation de la rotation de la caméra dans la méthode mouseMoved

```

bool InputListener::mouseMoved(const OIS::MouseEvent &e←
)
{
    mCamera->yaw(Ogre::Degree(-mVitesseRotation * e.←
        state.X.rel));
    mCamera->pitch(Ogre::Degree(-mVitesseRotation * e.←
        state.Y.rel));

    return true;
}

```

Vous pouvez maintenant compiler et exécuter votre application ; les commandes de déplacement sont maintenant gérées entièrement par notre classe

InputListener, n'hésitez donc pas à adapter les variables initialisées dans le constructeur si vous voulez accélérer ou ralentir les mouvements par exemple.

Cette méthode de gestion des entrées permet donc de gérer plus facilement l'appui ponctuel sur une touche, tout en conservant une gestion simple des touches qui peuvent rester enfoncées (pour le déplacement ici).

Gardez cependant bien à l'esprit qu'**il ne peut y avoir qu'un seul écouteur par périphérique d'entrée** et qu'il faudra donc penser à rapporter l'appui sur les touches à des écouteurs annexes lorsque votre application grossira, sinon vous allez vite vous retrouver avec un code lourd et mal organisé.

Dans ce chapitre, nous avons vu comment gérer nous-mêmes les entrées de l'utilisateur avec la bibliothèque OIS, ainsi que le principe des frame listeners, qui nous ont ici été bien utiles alors que nous n'avons pas encore vu le fonctionnement de la boucle de rendu.

Le prochain chapitre promet d'être intéressant : nous allons en effet utiliser le module de terrain d'Ogre qui a été refait dans la version 1.7 et qui offre une gestion beaucoup plus optimisée des terrains par rapport aux versions précédentes. Je n'en dis pas plus, on se retrouve de l'autre côté.

5.4.2 Code

Le CMakeLists.txt n'est pas modifié par rapport à la gestion des entrées sans buffer.

Listing 5.21: PremiereApplication::createFrameListener()

```
void PremiereApplication :: createFrameListener ()
{
    //activation du buffer pour la souris et le clavier
    mFrameListener = new InputListener(mWindow, mCamera
        , mSceneMgr, true, true, true);
    //root est l'element de base de l'application Ogre
    qui s'occupe notamment de gerer les frames
    listeners
    mRoot -> addFrameListener( mFrameListener );
}
```

Listing 5.22: InputListener.h

```
#include "ExampleFrameListener.h"

//nous utiliserons OIS pour gerer les entrees de l'
utilisateur
#include <OIS/OIS.h>

//la classe ExampleFrameListener ExampleApplication met
deja en oeuvre une gestion des entrees et attend un
ExampleFrameListener
```

```

//ExampleFrameListener s'occupe aussi de construire les ←
//objets nécessaires à l'écoute des entrées
class InputListener : public ExampleFrameListener, OIS::KeyListener, OIS::MouseListener
{
public:
    InputListener(RenderWindow* win, Camera* cam, ←
        SceneManager *sceneMgr, ←
        bool bufferedKeys = false, bool ←
        bufferedMouse = false, ←
        bool bufferedJoy = false);

    //nous allons juste devoir redéfinir ←
    //frameRenderingQueued() pour implementer ←
    //notre propre gestion des entrées à la place ←
    //de celle prevue par ExampleFrameListener.
    virtual bool frameRenderingQueued(const ←
        FrameEvent& evt);

    //les 5 méthodes suivantes sont des méthodes ←
    //virtuelles pures à reimplementer
    //ces méthodes seront appelées lors de l'événement ←
    //correspondant, leur valeur de retour ne dit ←
    //pas si on doit continuer ou pas, d'où l'←
    //intérêt de mContinuer
    bool mouseMoved(const OIS::MouseEvent &e); ←
        //evenmnt: la souris a bougé
    bool mousePressed(const OIS::MouseEvent &e, OIS::MouseButtonID id); //evenmnt: un bouton de ←
        //la souris a été pressé
    bool mouseReleased(const OIS::MouseEvent &e, ←
        OIS::MouseButtonID id); //evenmnt: un bouton ←
        //de la souris a été relâché
    bool keyPressed(const OIS::KeyEvent &e); //←
        //evenmnt: un bouton a été pressé
    bool keyReleased(const OIS::KeyEvent &e); //←
        //evenmnt: un bouton a été relâché

private:
    Ogre::SceneManager *mSceneMgr;
    Ogre::Camera *mCamera;

    //permettra d'enregistrer l'appui sur ←
    //ECHAP
    bool mContinuer;

    //vecteur de la direction ds lqll on doit ←
    //bouger
    Ogre::Vector3 mMouvement;
}

```

```

Ogre::Real mVitesse;
//pour ajuster la vitesse de rotation de la souris
Ogre::Real mVitesseRotation;
Ogre::Radian mRotationX;
Ogre::Radian mRotationY;
};

```

Listing 5.23: InputListener.cpp

```

#include <OIS/OIS.h>
#include "InputListener.h"

InputListener :: InputListener(RenderWindow* win, Camera* cam, SceneManager *sceneMgr,
                                bool bufferedKeys, bool bufferedMouse, bool bufferedJoy)
: ExampleFrameListener(win, cam, bufferedKeys, bufferedMouse, bufferedJoy)
{
    mCamera = cam;
    mSceneMgr = sceneMgr;

    mContinuer = true; //permettra l enregistrement de l appuie sur ECHAP
    mMouvement = Ogre::Vector3::ZERO; //sera le vct de la direction ds la qll on doit se déplacer

    mVitesse = 100;
    mVitesseRotation = 0.3; //fcteur multiplicatif pour ajuster la rotation de la camera

    //enregistrement d un écouteur pour la souris et pour le clavier
    mMouvement->setEventCallback(this);
    mKeyboard->setEventCallback(this);
}

bool InputListener :: frameRenderingQueued(const FrameEvent& evt)

```

```

{
    //ces lignes permettent la mise a jour de nos ←
    //objets pour obtenir le nom des touches enfoncées
    if(mMouse){
        mMouse->capture();
    }
    if(mKeyboard){
        mKeyboard->capture();
    }

    //gestion du mouvement de la camera
    Ogre::Vector3 deplacement = Ogre::Vector3::ZERO;
    deplacement = mMouvement * mVitesse * evt.←
        timeSinceLastFrame;
    mCamera->moveRelative(deplacement);

    return mContinuer;
}

//les 5 methodes suivantes sont des methodes virtuelles←
//pures a reimplementer
//si la souris a bouge
bool InputListener::mouseMoved(const OIS::MouseEvent &e←
)
{
    mCamera->yaw(Ogre::Degree(-mVitesseRotation * e.←
        state.X.rel));
    mCamera->pitch(Ogre::Degree(-mVitesseRotation * e.←
        state.Y.rel));

    return true;
}

//si une touche de la souris est pressee
bool InputListener::mousePressed(const OIS::MouseEvent &e, OIS::MouseButtonID id)
{
    return true;
}

//si une touche de la souris est relachee
bool InputListener::mouseReleased(const OIS::MouseEvent &e, OIS::MouseButtonID id)
{
    return true;
}

//si une touche est pressee

```

```

bool InputListener :: keyPressed(const OIS :: KeyEvent &e)
{
    switch(e.key)
    {
        case OIS :: KC_ESCAPE:
            mContinuer = false;
            break;
        case OIS :: KC_W:
            mMouvement.z -= 1;
            break;
        case OIS :: KC_S:
            mMouvement.z += 1;
            break;
        case OIS :: KC_A:
            mMouvement.x -= 1;
            break;
        case OIS :: KC_D:
            mMouvement.x += 1;
            break;
        case OIS :: KC_LSHIFT:
            mVitesse *= 2;
            break;
    }

    return true;
}

// si une touche est relachée
bool InputListener :: keyReleased(const OIS :: KeyEvent &e)
{
    switch(e.key)
    {
        case OIS :: KC_W:
            mMouvement.z += 1;
            break;
        case OIS :: KC_S:
            mMouvement.z -= 1;
            break;
        case OIS :: KC_A:
            mMouvement.x += 1;
            break;
        case OIS :: KC_D:
            mMouvement.x -= 1;
            break;
        case OIS :: KC_LSHIFT:
            mVitesse /= 2;
            break;
    }
}

```

```
    return true;  
}
```


Chapter 6

Gardez les pieds sur Terre

Plutôt que de placer notre terrain comme une simple entité dans la scène (ce qui serait très laborieux), Ogre propose de passer par une classe Terrain pour la gestion des terrains dans la scène.

Le terrain n'est généralement pas seul et le ciel joue un rôle important pour le réalisme de la scène. Là encore, quelques outils bienvenus offrent différentes solutions pour obtenir un résultat convaincant.

6.1 Créer un terrain

6.1.1 Préparation

Avant de commencer, il va falloir modifier un peu notre projet avec de nouvelles dépendances pour que les terrains soient utilisables.

Il faut ajouter un fichier en-tête dans notre classe :

Listing 6.1: Ajout du fichier d'entête pour la gestion des terrains

```
#include <Ogre/Terrain/OgreTerrain.h>
```

Pour la prise en compte des lib

- OgreTerrain.dll
- OgrePaging.dll

il va nous ajouter les lignes suivantes au fichier CMakeList.txt:

Listing 6.2: Modification de CMakeLists.txt pour l'inclusion des lib Terrain et Paging

```
#Inclusion de la lib libOgreTerrain.so
if (OGRE_Terrain_FOUND)
    set(OGRE_LIBRARIES ${OGRE_LIBRARIES} ${←
        OGRE_Terrain_LIBRARIES})
    message(STATUS "Found OGRE_Terrain: ${←
        OGRE_Terrain_LIBRARIES}")
else (OGRE_Terrain_FOUND)
    message(SEND_ERROR "OgreTerrain Library not found.")
```

```

endif(OGRE_Terrain_FOUND)

#Inclusion de la lib libOgrePaging.so
if (OGRE_Paging_FOUND)
    set(OGRE_LIBRARIES ${OGRE_LIBRARIES} ${←
        OGRE_Paging_LIBRARIES})
    message(STATUS "Found OGRE_Paging: ${←
        OGRE_Paging_LIBRARIES}")
else (OGRE_Paging_FOUND)
    message(SEND_ERROR "OgrePaging Library not found.")
endif(OGRE_Paging_FOUND)

```

6.1.2 Quelques paramètres à régler

Ajout d'attributs obligatoires pour le terrain

Commençons par ajouter deux attributs dans notre classe PremiereApplication.

- un objet Terrain, qui gérera les propriétés de notre terrain,
- un objet TerrainGlobalOptions, qui définit des propriétés générales pour les terrains dans notre application, notamment l'éclairage.

La présence de cet objet TerrainGlobalOptions est obligatoire lorsque vous voulez utiliser les terrains dans votre scène, sous peine d'erreur à l'exécution.

Listing 6.3: Attributs pour la gestion de terrain

```

Ogre::Terrain *mTerrain;
Ogre::TerrainGlobalOptions *mGlobals;
```

Au début de la méthode createScene(), nous allons régler quelques paramètres pour que notre terrain apparaisse sous son meilleur jour.

Réglage de la caméra

Premièrement, je vous conseille d'augmenter la distance de vue de la caméra et de la positionner en hauteur :

Listing 6.4: Réglage de la caméra

```

mCamera->setFarClipDistance(20000);
mCamera->setPosition(0, 500, 0);
```

Il est aussi possible de régler la distance de vue à l'infini, en mettant 0 comme paramètre. Cependant, cela dépend de votre machine, il faut donc vérifier si vous pouvez vous le permettre avant de l'appliquer.

Listing 6.5: Vérification et réglages de vue à l'infini

```

if (mRoot->getRenderSystem()->getCapabilities()->←
    hasCapability(Ogre::RSC_INFINITE_FAR_PLANE))
    mCamera->setFarClipDistance(0);
```

Définition des éclairages

Plutôt que de gérer l'éclairage de façon distincte, on peut définir un éclairage d'ambiance particulier pour le terrain. Pour cela, il suffit de définir une lumière directionnelle avec les paramètres qui vous paraissent adaptés à votre environnement.

Ici cette lumière est ajoutée en tant qu'attribut de la classe, car je l'utiliserais dans une autre fonction.

Listing 6.6: Définition de l'éclairage pour le terrain

```
Ogre::Vector3 lightdir (0.55f, -0.3f, 0.75f);
mLight = mSceneMgr->createLight ('terrainLight');
mLight->setType(Ogre::Light::LT_DIRECTIONAL);
mLight->setDirection (lightdir);
mLight->setDiffuseColour (Ogre::ColourValue::White);
mLight->setSpecularColour (Ogre::ColourValue(0.4f, 0.4f, ←
    0.4f));
```

Définition d'une méthode pour la prise en charge de l'initialisation du terrain

Nous allons définir une méthode `createTerrain()`, qui sera appelée dans `createScene()` et qui prendra en charge toute l'initialisation du terrain.

à l'intérieur de celle-ci, commencez par créer le `TerrainGlobalOptions` :

Listing 6.7: Méthode pour la prise en charge de l'initialisation du terrain

```
void PremiereApplication :: createTerrain ()
{
    mGlobals = OGRE_NEW Ogre::TerrainGlobalOptions();
    mGlobals->setMaxPixelError (8);
}
```

`mGlobals` est le premier objet d'`Ogre` que nous allons créer nous-mêmes, sans passer par l'intermédiaire du `Scene Manager`.

Le moteur fournit différentes macros comme `OGRE_NEW` pour allouer l'espace en mémoire lorsque vous instanciez une classe d'`Ogre`. De façon générale, les développeurs conseillent d'utiliser ces macros lorsque vous devez faire de l'allocation dynamique sur les objets du moteur qui dérivent de `Ogre::AllocatedObject`. Vous pouvez voir la hiérarchie des classes dans la documentation.

En ce qui concerne la destruction des objets, l'utilisation de l'opérateur `OGRE_NEW` implique l'utilisation de l'opérateur `OGRE_DELETE` pour libérer l'espace mémoire.

La seconde ligne appelle la méthode `setMaxPixelError()` qui donne la précision avec laquelle le terrain est rendu. La valeur indiquée est l'erreur tolérée, en pixels, pour l'affichage du terrain. Plus la valeur est faible, plus le terrain correspond au modèle donné, plus elle est forte et plus il sera imprécis, en donnant l'impression d'un terrain nivélisé.

On applique maintenant les réglages concernant l'éclairage à nos options globales.

Listing 6.8: Application des réglages

```
mGlobals->setLightMapDirection( mLight-><-
    getDerivedDirection() );
mGlobals->setCompositeMapDistance(3000);
mGlobals->setCompositeMapAmbient( mSceneMgr-><-
    getAmbientLight() );
mGlobals->setCompositeMapDiffuse( mLight-><-
    getDiffuseColour() );
```

6.1.3 Le terrain

Maintenant, passons à la création du terrain lui-même. Comme pour les options du terrain, nous ne passons pas par une méthode du Scene Manager pour créer le terrain, mais par le constructeur de la classe, qui prend tout de même en paramètre le Scene Manager de votre scène.

Listing 6.9: Création du terrain

```
mTerrain = OGRENEW Ogre::Terrain(mSceneMgr);
```

Il faut à présent définir précisément les paramètres de notre terrain :

- son relief,
- sa taille,
- ses textures.

Il est temps que je vous parle des heightmaps pour la modélisation du terrain !

Les heightmaps

Une heightmaps est une image qui contient des informations de relief. On les utilise pour stocker le relief d'un terrain, mais aussi pour connaître le relief d'une texture, afin de donner l'impression de relief sur un mesh alors qu'en réalité il est plat ou simplement lisse (c'est le principe du bump mapping).

Cette méthode a le principal avantage d'être très légère en terme de stockage, puisque l'on a simplement une image à la place d'un modèle 3D complet qui prendrait beaucoup plus d'espace à stocker.

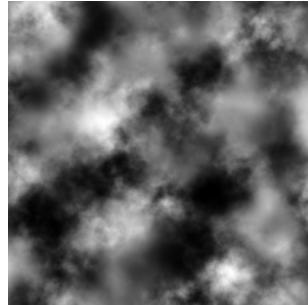
Voici l'image que nous allons utiliser pour notre terrain :

Comme vous le voyez c'est une simple image en noir et blanc, et pourtant cela suffit amplement !

En effet, si l'on utilise uniquement des niveaux de gris dans une image, chaque pixel peut prendre 256 valeurs, 0 correspondant au noir et à la hauteur la plus faible, 255 au blanc et à la plus forte altitude. On a donc 256 altitudes possibles pour notre terrain, ce qui est tout à fait honnête et suffit à la majorité des cas.

En réalité, chaque pixel possède trois valeurs, correspondant à la quantité de rouge, de vert et de bleu, chacune de ces valeurs allant de 0 à 255. Or pour les niveaux de gris, ces trois valeurs doivent être identiques, ce qui laisse 255

Figure 6.1: Terrain.png



triplets de valeurs: (0, 0, 0) pour le noir, puis les niveaux de gris et enfin (255, 255, 255) pour le blanc).

Lors de la création d'un fichier heightmap, on fait en sorte que le point le plus haut de notre carte soit blanc et que le point le plus bas soit noir, afin d'utiliser toute la plage de valeurs disponibles dans la carte et éviter les dénivellations peu naturelles.

Pour charger un fichier heightmap, on passe par un objet Image qui va chercher le nom du fichier que vous voulez dans les ressources déjà chargées. J'utilise le fichier terrain.png, que vous pouvez trouver dans "media/materials/textures".

Listing 6.10: Chargement du fichier heightmap

```
Ogre::Image img;
img.load("terrain.png", Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
```

Les paramètres géométriques

Pour fournir toutes les informations dont le terrain a besoin pour être généré, on utilise sa méthode prepare() qui prend en paramètre un Terrain::ImportData, qui est en gros une classe contenant l'ensemble des paramètres à fournir au terrain. On va donc commencer par créer cet objet :

Listing 6.11: Création de l'objet ImportData pour la définition des paramètres à fournir au terrain

```
Ogre::Terrain::ImportData imp;
imp.inputImage = &img; //recuperation de l'image
imp.terrainSize = img.getWidth(); //recuperation de la taille de l'image
imp.worldSize = 8000; //indique la taille du terrain
imp.inputScale = 600; //echelle adoptee pour l'altitude du terrain
imp.minBatchSize = 33; //taille min du batch pour le terrain
imp.maxBatchSize = 65; //taille max du batch pour le terrain
```

On commence par récupérer l'image et sa taille avec les lignes 2 et 3. **étant donné que les terrains sont carrés, votre image doit elle aussi être carrée**, faites attention à cela.

Ensuite, le paramètre worldSize indique la taille du terrain, c'est-à-dire la longueur de ses côtés en unités de la scène. Plus ce nombre est grand, plus l'image est agrandie.

inputScale correspond à l'échelle adoptée pour l'altitude du terrain. C'est la hauteur qui sépare un point de la carte représenté par un pixel noir d'un point représenté par un pixel blanc. Il doit donc être choisi en parallèle avec la taille du monde, puisque s'il est trop élevé et que le monde est trop petit, vous aurez un relief très escarpé.

Les deux dernières valeurs minBatchSize et maxBatchSize renseignent les tailles minimale et maximale de batch pour notre terrain.

La Batch Size

Le mot anglais batch signifie "lot" ou "paquet". L'affichage de modèle 3D à l'écran consommant beaucoup de ressources, plutôt que de chercher à calculer dans les moindres détails la façon dont apparaît la pelouse à l'autre bout du paysage, pour ensuite ne l'afficher que sur une toute petite surface de l'écran, le moteur va simplifier les choses et calculer de façon grossière l'affichage de ces objets.

Ainsi, les textures peuvent être simplifiées, mais aussi les meshs, dont l'ordinateur va réduire le nombre de vertices pour avoir moins de calculs à faire, vu que vous ne voyez pas les détails (on parle aussi de niveau de détail, ou LOD).

Pour un terrain, le maillage pourrait donc avoir un aspect similaire à celui-ci (image issue du wiki d'Ogre3D.org) : Image utilisateur

Le terrain est divisé en lots dont la taille varie en fonction de la distance de la caméra à ces lots. Plus on s'éloigne, plus le lot est simplifié par suppression de vertices. Lorsque plusieurs lots atteignent une taille minimale, ils sont regroupés en un seul lot, qui est à son tour simplifié progressivement si la caméra continue de reculer.

La zone où se situe la caméra est la plus détaillée, le reste est simplifié.

Si la taille minimum de batch est faible, les lots adjacents auront plus facilement un niveau de détail équivalent, mais il y aura plus de lots à gérer par l'ordinateur. En revanche, si elle est élevée, on regroupe plus rapidement les lots, mais les frontières entre ceux-ci sont plus facilement visibles, car le niveau de détail peut varier plus fortement.

Les valeurs que j'ai mises sont des valeurs courantes, sachez juste que la taille maximum est de 65 et qu'elles doivent obéir à la formule suivante : **taille=2n+1**

Mise en place des textures

Pour gérer les textures, l'outil Terrain d'Ogre utilise des calques. Chacun de ces calques correspond à une texture, que vous pourrez ensuite appliquer où bon vous semblera.

Comme on parle de calques, autant vous dire tout de suite qu'il est possible de les superposer, de donner plus ou moins d'intensité à un calque, pour créer des effets élaborés.

Nous allons commencer avec une seule texture pour faire simple et assimiler le principe. Tout se fait à l'aide de notre importateur de données :

Listing 6.12: Mise en place d'une texture pour le terrain

```
//donne la taille de la liste de calques
imp.layerList.resize(1);

//donne la taille de la texture dans le monde
imp.layerList[0].worldSize = 100;

//les deux lignes suivantes inserent chacune une ←
//texture dans notre calque
imp.layerList[0].textureNames.push\_back('grass\_green←
-01\_diffuseSpecular.dds','');
imp.layerList[0].textureNames.push\_back('grass\_green←
-01\_normalheight.dds','');
```

Ici les fonctions sont relativement explicites.

La première ligne donne la taille de la liste de calques, ici je n'en ai mis qu'un seul. La seconde ligne donne la taille de la texture dans le monde. **Plus le nombre¹ est important, plus la texture sera zoomée, et inversement.**

Les deux lignes suivantes insèrent chacune une texture dans notre calque (textureNames est un vector).

Deux textures ? Je croyais qu'on mettait les textures sur des calques différents ?

En fait, on devrait plutôt dire qu'un calque contient un matériau.

Les matériaux sont faits avec deux textures :

- une texture diffuse, qui contient les couleurs, les motifs du matériau ;
- une texture normale, contenant des informations sur le relief du matériau.

La combinaison de ces deux textures permet d'avoir un matériau complet.

Filtrage anisotrope

Je vais revenir rapidement sur le niveau de détail, qui est réglable pour les matériaux via le filtrage de texture.

Vous pouvez régler la netteté du placage de textures sur vos meshs via un niveau de filtrage.

On peut distinguer quatre options de filtrages, de la plus grossière à la plus précise :

- aucun filtrage ;
- bilinéaire ;
- trilinéaire ;
- anisotrope.

¹ce nombre est-il le nombre affecté à la taille de la texture?

Voici la différence entre un filtrage anisotrope (à gauche) et une texture sans filtrage (à droite). La différence est relativement subtile ici mais visible tout de même.

Image utilisateur

Je vous propose donc d'opter pour un filtrage anisotrope, avec une valeur de 8 (la valeur par défaut est 1 et équivaut à l'absence de filtrage). Vous devez donc rajouter ces deux lignes dans votre code, au début de la méthode `createScene()` par exemple.

Listing 6.13: Choix d'un filtrage anisotrope

```
Ogre::MaterialManager::getSingleton() .→
    setDefaultTextureFiltering(Ogre::TFO\ANISOTROPIC);
Ogre::MaterialManager::getSingleton() .→
    setDefaultAnisotropy(8);
```

Le filtrage de textures n'est pas propre uniquement aux terrains, mais affecte toutes les textures affichées par le moteur.

Chargement et nettoyage

Une fois que les paramètres ont été définis dans l'`ImportData`, il ne reste qu'à préparer et charger le terrain :

Listing 6.14: Préparation et chargement du terrain

```
mTerrain→prepare(imp);
mTerrain→load();
```

Pour terminer et faire un peu de place en mémoire, il est conseillé d'appeler la méthode suivante qui se chargera de libérer la mémoire allouée temporairement pour la création de votre terrain. Placez donc cette ligne à la fin de la méthode `createTerrain()`.

Listing 6.15: Libération de place en mémoire

```
mTerrain→freeTemporaryResources();
```

Compilez et lancez l'application pour obtenir un joli paysage !

6.2 Le plaquage de textures

Ogre nous permet d'utiliser différents calques pour nos matériaux. On va pouvoir mettre les calques les uns sur les autres, modifier leur opacité pour avoir une texture plus ou moins visible, tout en décidant de la zone où l'on veut appliquer la texture.

La première étape consiste à rajouter des calques dans notre liste, avant de créer le terrain. Remplacez le bloc que vous aviez par le code suivant, afin d'ajouter deux nouveaux matériaux.

Listing 6.16: Ajout de calques

```
imp.layerList.resize(3);
```

```

imp.layerList[0].worldSize = 100;
imp.layerList[0].textureNames.push_back("grass_green-01←
    _diffuse specular.dds");
imp.layerList[0].textureNames.push_back("grass_green-01←
    _normalheight.dds");

imp.layerList[1].worldSize = 30;
imp.layerList[1].textureNames.push_back("←
    growth_weirdfungus-03_diffuse specular.dds");
imp.layerList[1].textureNames.push_back("←
    growth_weirdfungus-03_normalheight.dds");

imp.layerList[2].worldSize = 200;
imp.layerList[2].textureNames.push_back("←
    dirt_grayrocky_diffuse specular.dds");
imp.layerList[2].textureNames.push_back("←
    dirt_grayrocky_normalheight.dds");

mTerrain->prepare(imp);
mTerrain->load();

```

Par défaut, c'est uniquement le premier matériau qui est affiché donc si vous exécutez le code maintenant, rien n'aura changé sur votre terrain. Il est donc nécessaire d'ajouter quelques lignes pour dire de quelle façon nous voulons faire notre plaquage de texture.

Le principe est simple à comprendre, chaque calque d'un terrain possède un objet `TerrainLayerBlendMap` (que j'appellerai dorénavant `Blend Map`) qui exprime la façon dont le calque est fusionné avec les calques inférieurs (le calque le plus bas est le calque 0).

Cette fusion est simplement une affaire de transparence. Les calques sont placés les uns sur les autres, et la composante transparente indique si la texture en dessous est plus ou moins visible. Il est de plus possible de faire varier la transparence du calque en chaque point de celui-ci, ce qui permet d'avoir un placage par zone.

à la suite du bloc de code précédent, commencez par récupérer les `Blend Map` correspondant au calque numéro 1, que nous venons d'ajouter (nous nous occuperons du second ensuite).

Listing 6.17: Récupération du Blend Map pour le premier terrain

```

Ogre::TerrainLayerBlendMap* blendMap1 = mTerrain->←
    getLayerBlendMap(1);

```

Nous allons commencer par plaquer une seule texture au-dessus de l'herbe, sur toute la surface de notre terrain.

L'idée est de parcourir l'ensemble des points du calque avec deux boucles `for` (il y a autant de points qu'il y avait de pixels dans notre `heightmap`) et de leur attribuer la transparence désirée, entre 1 (totalement opaque) et 255 (transparent).

La valeur la plus faible est bien 1 et non 0. Si vous mettez 0, la texture est transparente !

Listing 6.18: Attribution de la transparence désirée sur tous les points du calque

```
float* pBlend1 = blendMap1->getBlendPointer();

for (Ogre::uint16 y = 0; y < mTerrain->getLayerBlendMapSize(); ++y)
{
    for (Ogre::uint16 x = 0; x < mTerrain->getLayerBlendMapSize(); ++x)
    {
        // opacite desiree pour le point courant
        *pBlend1++ = 150;
    }
}
```

Pour terminer, il faut mettre à jour notre Blend Map en appelant les méthodes `dirty()` puis `update()`. La première sert à préciser que les données de la Blend map sont obsolètes et doivent être mises à jour, tandis que la seconde fait effectivement la mise à jour.

Si vous n'appelez pas d'abord `dirty()`, `update()` n'aura aucun effet.

Listing 6.19: Mise à jour de la Blend Map

```
blendMap1->dirty();
blendMap1->update();
```

Je vous mets le code complet de la méthode `createTerrain()` si vous voulez vérifier que tout est en ordre :

Listing 6.20: `createTerrain` (code complet)

```
mTerrain = OGRENEW Ogre::Terrain(mSceneMgr);

// options globales
mGlobals = OGRENEW Ogre::TerrainGlobalOptions();
mGlobals->setMaxPixelError(10);
mGlobals->setCompositeMapDistance(8000);
mGlobals->setLightMapDirection(mLight->getDerivedDirection());
Globals->setCompositeMapAmbient(mSceneMgr->getAmbientLight());
mGlobals->setCompositeMapDiffuse(mLight->getDiffuseColour());
Ogre::Image img;
img.load("terrain.png", Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);

// informations geometriques
Ogre::Terrain::ImportData imp;
imp.inputImage = &img;
```

```

imp.terrainSize = img.getWidth();
imp.worldSize = 8000;
imp.inputScale = 600;
imp.minBatchSize = 33;
imp.maxBatchSize = 65;

// textures
imp.layerList.resize(3);
imp.layerList[0].worldSize = 100;
imp.layerList[0].textureNames.push_back("grass_green-01←
    _diffuse specular.dds");
imp.layerList[0].textureNames.push_back("grass_green-01←
    _normalheight.dds");
imp.layerList[1].worldSize = 30;
imp.layerList[1].textureNames.push_back("←
    growth_weirdfungus-03_diffuse specular.dds");
imp.layerList[1].textureNames.push_back("←
    growth_weirdfungus-03_normalheight.dds");
imp.layerList[2].worldSize = 200;
imp.layerList[2].textureNames.push_back("←
    dirt_grayrocky_diffuse specular.dds");
imp.layerList[2].textureNames.push_back("←
    dirt_grayrocky_normalheight.dds");
mTerrain->prepare(imp);35 mTerrain->load();

// plaquage de texture
Ogre::TerrainLayerBlendMap* blendMap1 = mTerrain->←
    getLayerBlendMap(1);
float* pBlend1 = blendMap1->getBlendPointer();
for (Ogre::uint16 y = 0; y < mTerrain->←
    getLayerBlendMapSize(); ++y)
{
    for (Ogre::uint16 x = 0; x < mTerrain->←
        getLayerBlendMapSize(); ++x)
    {
        *pBlend1++ = 150;
    }
}

blendMap1->dirty();
blendMap1->update();

mTerrain->freeTemporaryResources();

```

Vous pouvez maintenant exécuter l'application ! Profitez-en pour vérifier en vous approchant du sol que l'on distingue bien l'herbe et par-dessus la terre.
Image utilisateur

On va tout de suite essayer de faire quelque chose de plus esthétique.

Pour cela, maintenant que vous avez saisi les grandes étapes, je vous propose un mini-TP pour vous entraîner.

6.2.1 Au boulot !

Objectif

Une image vaut sûrement mieux qu'un long discours, voici donc ce que vous allez devoir obtenir : Image utilisateur

L'idée est de texturer le terrain en fonction de l'altitude. Si l'on est en dessous d'un certain seuil, il n'y a que de la terre qui apparaît ; au-dessus, on a de l'herbe.

Indications

Je vous donne tout de même les méthodes qui sont utiles, notamment pour trouver l'altitude du terrain en fonction de la position :

Listing 6.21: méthode getHeightAtTerrainPosition pour trouver l'altitude du terrain en fonction de la position

```
float Ogre::Terrain::getHeightAtTerrainPosition(Ogre::Real x, Ogre::Real y)
```

Cette fonction retourne l'altitude en fonction de la position, sachant que x et y sont compris entre 0 et 1.

Pour récupérer ces deux valeurs, on utilise une méthode de TerrainLayerBlendMap qui convertit les coordonnées de l'image en coordonnées du terrain (celles dont vous avez besoin) :

Listing 6.22: Méthode convertImageToTerrainSpace pour convertir les coordonnées de l'image en coordonnées du terrain

```
void Ogre::TerrainLayerBlendMap::convertImageToTerrainSpace(size_t x, size_t y, Ogre::Real * outX, Ogre::Real * outY)
```

Enfin, je vous laisse choisir l'altitude limite entre l'herbe et la terre. Vous avez toutes les cartes en mains maintenant.

Correction

Comme avez dû le deviner, tout se passe dans les deux boucles for.

Pour chaque point parcouru, on recherche ses coordonnées dans le repère du terrain, puis on récupère la hauteur, que l'on compare à notre hauteur limite. Si on est en dessous, on affiche la texture de terre avec une opacité maximum, sinon on ne fait qu'incrémenter le pointeur de la Blend map.

Voici donc le code modifié :

Listing 6.23: Attribution de la transparence désirée sur tous les points du calque selon leur position

```
for (Ogre::uint16 y = 0; y < mTerrain->getLayerBlendMapSize(); ++y)
{
    for (Ogre::uint16 x = 0; x < mTerrain->getLayerBlendMapSize(); ++x)
    {
        // Code pour attribuer la transparence
    }
}
```

```

Ogre::Real terrainX, terrainY;
blendMap1->convertImageToTerrainSpace(x, y, &←
    terrainX, &terrainY);
Ogre::Real height = mTerrain->←
    getHeightAtTerrainPosition(terrainX, ←
        terrainY);
if(height < 200)
    *pBlend1 = 1;
pBlend1++;
}
}

```

Vous pouvez aussi bien sûr récupérer la Blend map numéro 2 plus haut pour voir le résultat avec une autre texture, c'est ce que j'ai fait pour obtenir l'image référence pour le TP.

Code

Ci-dessous le trouve le code de la metohde createTerrain et le resultat obtenu:

Listing 6.24: createTerrain

```

void PremiereApplication :: createTerrain()
{
    Ogre::Vector3 lightdir (0.55f, -0.3f, 0.75f);

    mLight = mSceneMgr->createLight("terrainLight");
    mLight->setType(Light :: LT_DIRECTIONAL);
    mLight->setDirection(lightdir);
    mLight->setDiffuseColour(Ogre::ColourValue :: White);
    mLight->setSpecularColour(Ogre::ColourValue (0.4f, ←
        0.4f, 0.4f));

    mGlobals = OGRENEW Ogre::TerrainGlobalOptions();
    mGlobals->setMaxPixelError(8); /*<specifie la ←
        precision de rendu (en nbr de pixel) du terrain←
    */

    //application des reglages de lumiere aux options ←
    //globales
    mGlobals->setLightMapDirection(mLight->←
        getDerivedDirection());
    mGlobals->setCompositeMapDistance(3000);
    mGlobals->setCompositeMapAmbient(mSceneMgr->←
        getAmbientLight());
    mGlobals->setCompositeMapDiffuse(mLight->←
        getDiffuseColour());

    mTerrain = OGRENEW Ogre::Terrain(mSceneMgr); /*<←
    Creation du terrain*/
}

```

```

Ogre::Image img;
img.load("terrain.png", Ogre::ResourceGroupManager::
    ::DEFAULT_RESOURCE_GROUP_NAME); /*<Chargement de<
    l'image Heighmaps contenant les infos pour le <
    relief*/
```

//Mise en place des proprietes generales du <
 terrain

```

Ogre::Terrain::ImportData imp; /*<creation de l'<
    objet qui contiendra toutes les infos pour le <
    terrain*/
imp.inputImage = &img; /*<recuperation de l'image*/
imp.terrainSize = img.getWidth(); /*<recuperation <
    de la taille de l'image (les terrains sont <
    carres, l'image doit donc etre carree)*/
```

```

imp.worldSize = 8000; /*<taille du terrain en <
    unites de la scene (plus ce nbre est grande, <
    plus l'image est grande)*/
imp.inputScale = 600; /*<echelle adoptee pour l'<
    altitude du terrain (difference entre un pt a l'<
    altitude 0 (pt en blanc ds terrain.png) et un <
    point a l'altitude max (pt en noir))*/
imp.minBatchSize = 33; /*<taille min du batch pour <
    le terrain*/
imp.maxBatchSize = 65; /*<taille max du batch pour <
    le terrain*/
```

//Mise en place des textures

```

imp.layerList.resize(1); /*<donne la taille de la <
    liste des calques*/
imp.layerList[0].worldSize = 100; /*<donne la <
    taille de la texture ds le monde (plus le nbre <
    est gd plus la texture est zoomee)*/
//insertion de textures dans notre calque
imp.layerList[0].textureNames.push_back("←
    grass-green-01_diffuseSpecular.dds"); /*>←
    contient les couleurs, les motifs du materiau*/
imp.layerList[0].textureNames.push_back("←
    grass-green-01_normalheight.dds"); /*<contient <
    des infos sur le relief du materiau*/
```

//Filtrage anisotropique (affecte toutes les <
 textures)

```

Ogre::MaterialManager::getSingleton().←
    setDefaultTextureFiltering(Ogre::TFO_ANISOTROPIC←
    );
Ogre::MaterialManager::getSingleton().←
    setDefaultAnisotropy(8);
```

```

///Rajout de calques dans notre liste
imp.layerList.resize(3);
imp.layerList[0].worldSize = 100;
imp.layerList[0].textureNames.push_back("←
    grass_green-01_diffusespecular.dds");
imp.layerList[0].textureNames.push_back("←
    grass_green-01_normalheight.dds");
imp.layerList[1].worldSize = 30;
imp.layerList[1].textureNames.push_back("←
    growth_weirdfungus-03_diffusespecular.dds");
imp.layerList[1].textureNames.push_back("←
    growth_weirdfungus-03_normalheight.dds");
imp.layerList[2].worldSize = 200;
imp.layerList[2].textureNames.push_back("←
    dirt_grayrocky_diffusespecular.dds");
imp.layerList[2].textureNames.push_back("←
    dirt_grayrocky_normalheight.dds");

mTerrain->prepare(imp);/**<on fournit toutes les ←
    informations nécessaires au terrain*/
mTerrain->load();/**<chargement du terrain*/

///recuperation du blend map pour le premier ←
    terrain
Ogre::TerrainLayerBlendMap* blendMap2 = mTerrain->←
    getLayerBlendMap(2);/**<recuperation du blend ←
    map*/


///plaquage de la texture..
float* pBlend2 = blendMap2->getBlendPointer();
///..en fonction de l'altitude
for (Ogre::uint16 y = 0; y < mTerrain->←
    getLayerBlendMapSize(); ++y)
{
    for (Ogre::uint16 x = 0; x < mTerrain->←
        getLayerBlendMapSize(); ++x)
    {
        Ogre::Real terrainX, terrainY;

        blendMap2->convertImageToTerrainSpace(x, y, ←
            &terrainX, &terrainY);
        Ogre::Real height = mTerrain->←
            getHeightAtTerrainPosition(terrainX, ←
                terrainY);

        //en general l'herbe se trouve en basse ←
            altitude
        if(height > 200)
            *pBlend2 = 1;
    }
}

```

```

        pBlend2++;
    }
}

blendMap2->dirty(); /*<precise que les donnees de la blend Map st obsoletes*/
blendMap2->update(); /*<fait la mise a jour des donnees de la blend map*/

mTerrain->freeTemporaryResources(); /*<liberation de la memoire allouee temporairement*/
}

```

Listing 6.25: PremiereApplication.h

```

using namespace std;

#include <ExampleApplication.h>
#include <OgreMovableObject.h>
#include "InputListener.h"

#include <Terrain/OgreTerrain.h>

class PremiereApplication : public ExampleApplication
{
public:
    Terrain *mTerrain;                                /**<-
        gerera les proprietes de notre terrain*/
    Ogre::TerrainGlobalOptions *mGlobals;             /**<-
        objet obligatoire pr la gestion des terrains<-
        , definira les proprietes generales pour le <-
        terrain (ex: eclairage)*/
    Light *mLight;                                    /**<-
        contiendra les prorietes de la lumiere*/
    /* Cree la scene avec une tete d'Ogre flottant<-
       sur un carre de pelouse
    */
    void createScene();

    /* Cree et place la camera
     */
    void createCamera();

    /* Cree une seconde vue de la scene
     */
    void createViewports();

    /* Gere les entrees
     */
}

```

```

/*
void createFrameListener();

/** Cree une lumiere selon le parametre
@param luxType - type de lumiere desiree
@param lux - instance de lumiere
@return distance to point
*/
//void createLux(std::string luxType, ←
    MovableObject *lux);

/** Initialise le terrain
*/
void createTerrain();
};
```

Listing 6.26: CMakeLists.txt

```

project(helloworld)
cmake_minimum_required(VERSION 2.6)

set(CMAKEMODULEPATH "/usr/share/OGRE/cmake/modules")

set(CMAKE_CXX_FLAGS "-Wall -std=c++11 -g")

# Il s'agit du tutoriel d'exemple, qui utilise quelques←
# fichiers prdfini de Ogre. Il faut indiquer a cmake ←
# ou se trouvent les includes en question
include_directories ("include")

# Bien sur, pour compiler Ogre, il faut le chercher, et←
# dfinir le rpertoire contenant les includes.
find_package(OGRE REQUIRED)
include_directories (${OGRE_INCLUDE_DIRS})

if (OGRE_Overlay_FOUND)
# append OgreOverlay to the end of the OGRE_LIBRARIES ←
# variable
set(OGRE_LIBRARIES ${OGRE_LIBRARIES} ${←
    OGRE_Overlay_LIBRARIES})
message(STATUS "Found OGRE_Terrain: ${←
    OGRE_Overlay_LIBRARIES}")
else (OGRE_Overlay_FOUND)
    message(SEND_ERROR "OgreOverlay Library not found.")
endif(OGRE_Overlay_FOUND)

#Inclusion de la lib libOgreTerrain.so
if (OGRE_Terrain_FOUND)
    set(OGRE_LIBRARIES ${OGRE_LIBRARIES} ${←
        OGRE_Terrain_LIBRARIES})
```

```

message(STATUS "Found OGRE_Terrain: ${OGRE_Terrain_LIBRARIES}")
else (OGRE_Terrain_FOUND)
    message(SEND_ERROR "OgreTerrain Library not found.")
endif(OGRE_Terrain_FOUND)

# Inclusion de la lib libOgrePaging.so
if (OGRE_Paging_FOUND)
    set(OGRE_LIBRARIES ${OGRE_LIBRARIES} ${OGRE_Paging_LIBRARIES})
    message(STATUS "Found OGRE_Paging: ${OGRE_Paging_LIBRARIES}")
else (OGRE_Paging_FOUND)
    message(SEND_ERROR "OgrePaging Library not found.")
endif(OGRE_Paging_FOUND)

# L'exemple depend aussi de OIS, une lib pour gerer la souris, clavier, joystick...
find_package(OIS REQUIRED)

# On definit les sources qu'on veut compiler
SET( SOURCES
    src/InputListener.cpp
    src/PremiereApplication.cpp
    src/main.cpp
)
# On les compile
add_executable (
    premiereapp ${SOURCES}
)
message(STATUS "OGRE Libraries used to link are ${OGRE_LIBRARIES}")

# Et pour finir, on lie l'executable avec les librairies que find_package nous a gentillement trouve.
target_link_libraries(premiereapp ${OGRE_LIBRARIES} ${OIS_LIBRARY} -lboost_system)

set( RESOURCES_FILE
    media/
    resources/ogre.cfg
    resources/plugins.cfg
    resources/resources.cfg
)

```

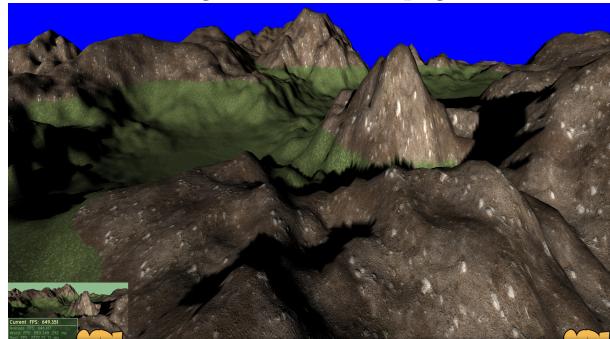
```

)
# do the copying
foreach( file_i ${RESOURCES_FILE} )
    add_custom_command(
        TARGET premiereapp
        POST_BUILD
        COMMAND cp -R ${CMAKE_SOURCE_DIR}/${file_i} ${←
            CMAKE_BINARY_DIR}
        COMMENT "copy file ${file_i}"
    )
endforeach( file_i )

```

Et on obtiend:

Figure 6.2: Terrain.png



6.2.2 Pour aller plus loin

Sur le même principe, nous allons voir comment appliquer deux textures différentes sur une petite largeur, à une altitude donnée.

Ceci peut être utile si vous voulez réaliser des étendues d'eau dans votre terrain : au bord de l'eau, il y a de la boue, un peu au-dessus, de la terre sèche, puis ensuite l'herbe reprend ses droits. C'est ce que nous allons faire, avec une opacité progressive, mais sans l'eau, ce sera pour plus tard.

Nous devons commencer par récupérer un pointeur sur notre seconde Blend Map pour pouvoir gérer la seconde texture en plus de la première. Il y a donc deux lignes à rajouter en conséquence avant les boucles.

Listing 6.27: Récupération des Blend Map pour le premier et le second terrain

```
Ogre::TerrainLayerBlendMap* blendMap1 = mTerrain->getLayerBlendMap(1);
Ogre::TerrainLayerBlendMap* blendMap2 = mTerrain->getLayerBlendMap(2);

float* pBlend1 = blendMap1->getBlendPointer();
float* pBlend2 = blendMap2->getBlendPointer();
```

Définissons aussi deux variables pour chacune des textures : la hauteur à laquelle se situe la texture et la largeur de la bande que l'on veut obtenir.

```
Ogre::Real minHeight1 = 70;
Ogre::Real fadeDist1 = 40;
Ogre::Real minHeight2 = 70;
Ogre::Real fadeDist2 = 15;
```

Dans la boucle, on déclare trois variables : deux coordonnées du terrain et la transparence pour le point actuel.

```
Ogre::Real terrainX, terrainY, transparence;
```

On récupère ensuite la hauteur du terrain comme précédemment.

```
blendMap1->convertImageToTerrainSpace(x, y, &terrainX, ←
    &terrainY);
Ogre::Real height = mTerrain->←
    getHeightAtTerrainPosition(terrainX, terrainY);
```

Ensuite, pour chaque texture, on calcule la différence entre la hauteur du point actuel et la hauteur que l'on veut pour la texture, divisée par la largeur de la bande. Si le point est censé être recouvert par la texture, ce nombre sera donc compris entre 0 et 1.

On utilise ensuite la méthode statique Clamp() qui a pour prototype :

Listing 6.28: Utilisation de la méthode statique Clamp()

```
static T Ogre::Math::Clamp(T val, T minval, T maxval)
```

Si val est inférieure à minval, la fonction retourne minval ; si val est supérieure à maxval, on retourne maxval. Si val est dans l'intervalle, on la retourne directement.

Comme on ne veut afficher que les points dont la valeur calculée précédemment est comprise entre 0 et 1, on va utiliser cette méthode pour "couper" toutes les valeurs en dehors de l'intervalle.

```
transparence = (height - minHeight1) / fadeDist1;
transparence = Ogre::Math::Clamp(transparence, (Ogre::←
    Real) 0, (Ogre::Real) 1);
```

Pour terminer, on multiplie transparence par 255 pour avoir une valeur comprise entre 0 et 255.

```
*pBlend1++ = transparence * 255;
```

On observe que si transparence est à l'extérieur de l'intervalle [0 ; 1] après le premier calcul, Clamp retournera 0 ou 1. Quand on multiplie par 255, on obtient donc 0 ou 255, qui sont les deux valeurs pour lesquelles la texture est transparente. Mission accomplie !

On copie ces trois lignes pour la seconde texture, et on obtient le code suivant dans nos boucles :

```

for (Ogre::uint16 y = 0; y < mTerrain-><b>getLayerBlendMapSize()</b>; ++y)
{
    for (Ogre::uint16 x = 0; x < mTerrain-><b>getLayerBlendMapSize()</b>; ++x)
    {
        Ogre::Real terrainX, terrainY, transparence;
        blendMap1->convertImageToTerrainSpace(x, y, &terrainX, &terrainY);
        Ogre::Real height = mTerrain-><b>getHeightAtTerrainPosition(terrainX, terrainY)</b>;
        transparence = (height - minHeight1) / fadeDist1;
        transparence = Ogre::Math::Clamp(transparence, (Ogre::Real)0, (Ogre::Real)1);
        *pBlend1++ = transparence * 255;
        transparence = (height - minHeight2) / fadeDist2;
        transparence = Ogre::Math::Clamp(transparence, (Ogre::Real)0, (Ogre::Real)1);
        *pBlend2++ = transparence * 255;
    }
}

```

Pensez à mettre à jour la seconde Blend Map une fois que les modifications sont terminées :

```

blendMap1->dirty();
blendMap2->dirty();
blendMap1->update();
blendMap2->update();

```

Code

Les seules choses à changer sont le plaquage de texture:

Listing 6.29: Plaquage de textures sur zone

```

...
    //pour definir des bandes d'altitude avec une texture differente
    Ogre::Real minHeight1 = 70;
    Ogre::Real fadeDist1 = 40;
    Ogre::Real minHeight2 = 70;
    Ogre::Real fadeDist2 = 15;

    //recuperation du blend map pour le premier terrain

```

```

Ogre::TerrainLayerBlendMap* blendMap1 = mTerrain->↔
    getLayerBlendMap(1); /*<recuperation du blend map* /
Ogre::TerrainLayerBlendMap* blendMap2 = mTerrain->↔
    getLayerBlendMap(2); /*<recuperation du blend map* /

///plaque de la texture..
float* pBlend1 = blendMap1->getBlendPointer();
float* pBlend2 = blendMap2->getBlendPointer();
///..en fonction de l'altitude
for (Ogre::uint16 y = 0; y < mTerrain->↔
    getLayerBlendMapSize(); ++y)
{
    for (Ogre::uint16 x = 0; x < mTerrain->↔
        getLayerBlendMapSize(); ++x)
    {
        Ogre::Real terrainX, terrainY, transparence↔
            ;
        blendMap1->convertImageToTerrainSpace(x, y, ↔
            &terrainX, &terrainY);
        Ogre::Real height = mTerrain->↔
            getHeightAtTerrainPosition(terrainX, ↔
            terrainY);

        transparence = (height - minHeight1) / ↔
            fadeDist1;
        transparence = Ogre::Math::Clamp(↔
            transparence, (Ogre::Real)0, (Ogre::Real)1);
        *pBlend1++ = transparence * 255;

        transparence = (height - minHeight2) / ↔
            fadeDist2;
        transparence = Ogre::Math::Clamp(↔
            transparence, (Ogre::Real)0, (Ogre::Real)1);
        *pBlend2++ = transparence * 255;
    }
}

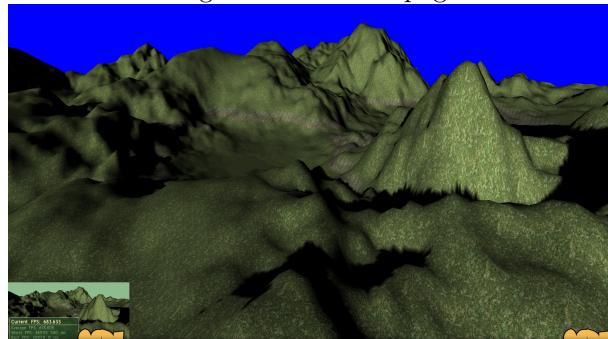
blendMap1->dirty(); /*<precise que les donnees de la blend Map st obsoletes* /
blendMap1->update(); /*<fait la mise a jour des donnees de la blend map* /
blendMap2->dirty(); /*<precise que les donnees de la blend Map st obsoletes* /
blendMap2->update(); /*<fait la mise a jour des donnees de la blend map* /

```

```
...
```

Et on obtient:

Figure 6.3: Terrain.png



6.3 Les groupes de terrains

Un groupe de terrains (ou TerrainGroup) range les terrains comme dans un tableau à deux dimensions. Dans le groupe, tous les terrains doivent avoir la même taille, afin de pouvoir les aligner dans une grille.

6.3.1 Crédation

On commence par ajouter un include au début de la classe puis on remplace notre instance de Terrain par un TerrainGroup, ensuite on l'initialisera dans notre méthode createTerrain() :

Listing 6.30: TerrainGroup: include et création

```
#include <Ogre/Terrain/OgreTerrainGroup.h>
//...
Ogre::TerrainGroup *mTerrainGroup;
```

Après les lignes permettant de charger l'image heightmap dans la méthode createTerrain(), insérez les lignes suivantes.

```
mTerrainGroup = OGRE_NEW Ogre::TerrainGroup(mSceneMgr, ←
    Ogre::Terrain::ALIGN_X_Z, img.getWidth(), 8000); ←

//On définit ensuite la position de l'origine du groupe ←
//de terrains
mTerrainGroup->setOrigin(Ogre::Vector3::ZERO); ←

//nom (et l'extension) que l'on veut attribuer à nos ←
//fichiers qui seront créés pour sauvegarder les ←
//terrains
mTerrainGroup->setFilenameConvention(Ogre::String("←
    TerrainDuZero"), Ogre::String("dat")); ←
```

Les paramètres à fournir au TerrainGroup sont

- le Scene manager,
- l’alignement du terrain par rapport au repère global (vous pouvez créer un terrain vertical, par exemple),
- la taille des heightmaps utilisées,
- la taille d’un terrain.

On définit ensuite la position de l’origine du groupe de terrains, puis le nom (et l’extension) que l’on veut attribuer à nos fichiers qui seront créés pour sauvegarder les terrains par la suite.

Pour les données des terrains enregistrées dans un objet ImportData, nous allons modifier un peu le fonctionnement du programme. On récupère en fait directement une référence sur un ImportData fourni par le TerrainGroup, que l’on modifie directement et qui sera valable pour l’ensemble des terrains du groupe. à noter que la ligne de définition de la heightmap n’est plus utile ici, cela sera indiqué lors de la création des terrains.

```
Ogre::Terrain ::ImportData& imp = mTerrainGroup-><-
    getDefaultImportSettings();
imp.terrainSize = img.getWidth();
imp.worldSize = 8000;
imp.inputScale = 600;
imp.minBatchSize = 33;
imp.maxBatchSize = 65;
```

Il est maintenant temps de créer les terrains du groupe. On définit la taille du groupe et pour chaque case, on appelle une méthode definirTerrain() définie plus bas qui s’occupera de créer chaque terrain indépendamment.

Listing 6.31: Création des terrains du groupe

```
int largeur = 2, longueur = 2;

for(int x = 0; x < largeur; x++)
{
    for(int y = 0; y < longueur; y++)
    {
        definirTerrain(x, y);
    }
}
mTerrainGroup->loadAllTerrains(true);
```

Le groupe se charge pour terminer d’appeler les méthodes load() de chaque terrain à travers la méthode loadAllTerrains(). Cette méthode prend un booléen en paramètre qui indique si le chargement doit être synchrone, c'est-à-dire exécuté dans un seul thread (le thread principal ici). Par défaut cette valeur est fausse, c'est-à-dire que les terrains sont chargés dans plusieurs threads si c'est possible.

Le chargement devient vite très lourd si l’on ajoute beaucoup de terrains aux groupes. Nous verrons plus bas comment accélérer le chargement.

Maintenant, nous devons écrire la méthode `definirTerrain()` qui utilise la méthode `defineTerrain()` de `TerrainGroup`. Celle-ci va prendre 3 paramètres : les deux coordonnées du terrain dans le groupe de terrains (sa position sur la grille, donc) et l'image `heightmap` utilisée pour ce terrain. Les coordonnées du terrain au sein du groupe peuvent être négatives.

Juste avant d'appeler le terrain, on va faire une vérification sur les coordonnées : si l'abscisse du terrain est impaire, on inverse l'image suivant l'axe Y, si l'ordonnée est impaire, on inverse l'image cette fois-ci selon l'axe X. Cela permet aux terrains du groupe de ne pas avoir de différence d'altitude lors des jointures. Si vous utilisez des `heightmaps` différents sur les terrains du groupe (ce qui sera probablement le cas), vous devrez faire attention à ce que les altitudes des bords correspondent pour éviter les trous à ces endroits.

Listing 6.32: `PremiereApplication.definirTerrain`

```
void PremiereApplication :: definirTerrain (int x, int y)
{
    Ogre::Image img;
    img.load ("terrain.png", Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);

    if(x % 2 != 0)
        img.flipAroundY();

    if(y % 2 != 0)
        img.flipAroundX();

    mTerrainGroup->defineTerrain(x, y, &img);
}
```

Une fois que les terrains sont chargés, il faut leur appliquer les textures définies. On utilise pour cela un itérateur sur le groupe de terrains et, pour chaque terrain, on appelle une méthode `initBlendMaps()` qui contient le code pour texturer les terrains.

```
Ogre::TerrainGroup::TerrainIterator ti = mTerrainGroup->getTerrainIterator();

while( ti.hasMoreElements() )
{
    Ogre::Terrain* t = ti.getNext()->instance;
    initBlendMaps( t );
}
```

La méthode `initBlendMaps()` contient uniquement du code que l'on a déjà vu mais que j'ai déplacé pour plus de clarté. Elle prend en paramètre le terrain dont on doit modifier les Blend maps.

```
void PremiereApplication :: initBlendMaps (Ogre::Terrain *terrain)
{
    Ogre::TerrainLayerBlendMap* blendMap1 = terrain->getLayerBlendMap(1);
```

```

Ogre::TerrainLayerBlendMap* blendMap2 = terrain->-
    getLayerBlendMap(2);
Ogre::Real minHeight1 = 70;
Ogre::Real fadeDist1 = 40;
Ogre::Real minHeight2 = 70;
Ogre::Real fadeDist2 = 15;
float* pBlend1 = blendMap1->getBlendPointer();
float* pBlend2 = blendMap2->getBlendPointer();

for (Ogre::uint16 y = 0; y < terrain->-
    getLayerBlendMapSize(); ++y)
{
    for (Ogre::uint16 x = 0; x < terrain->-
        getLayerBlendMapSize(); ++x)
    {
        Ogre::Real terrainX, terrainY, transparence-
            ;
        blendMap1->convertImageToTerrainSpace(x, y,-
            &terrainX, &terrainY);
        Ogre::Real height = terrain->-
            getHeightAtTerrainPosition(terrainX, -
                terrainY);
        transparence = (height - minHeight1) / -
            fadeDist1;
        transparence = Ogre::Math::Clamp(-
            transparence, (Ogre::Real)0, (Ogre::Real)-
            1);
        *pBlend1++ = transparence * 255;
        transparence = (height - minHeight2) / -
            fadeDist2;
        transparence = Ogre::Math::Clamp(-
            transparence, (Ogre::Real)0, (Ogre::Real)-
            1);
        *pBlend2++ = transparence * 255;
    }
}
blendMap1->dirty();
blendMap2->dirty();
blendMap1->update();
blendMap2->update();
}

```

Pour terminer, comme avec un terrain seul, on libère la mémoire utilisée par le TerrainGroup.

```
mTerrainGroup->freeTemporaryResources();
```

Votre scène doit maintenant avoir une surface plus grande que la première fois (on a maintenant quatre terrains). Vous pouvez encore augmenter le nombre de terrains, mais attention, le temps de chargement augmente rapidement !

6.3.2 Optimiser le temps de chargement

Vous avez certainement remarqué que la génération du terrain prend un temps conséquent lorsque le groupe s'agrandit. La création du terrain à partir du fichier heightmap nécessite en effet de convertir les données de l'image en données exploitables par le moteur.

Afin de réduire le temps de chargement, il est possible d'enregistrer un fichier qui contient toutes les informations sur le terrain construit pour éviter de relire l'image à chaque lancement de l'application. L'inconvénient majeur est la place occupée par ces fichiers générés, qui contiennent beaucoup plus d'informations qu'une simple heightmap.

En regardant la création du groupe de terrains, vous voyez que l'on a défini une convention de nommage pour des fichiers, mais qui est pour l'instant inutilisée.

```
mTerrainGroup->setFilenameConvention(Ogre::String("TerrainDuZero"), Ogre::String("dat"));
```

Cette ligne sert lors de la sauvegarde de fichiers de terrain : ceux-ci seront nommés en commençant par "TerrainDuZero" suivi d'un nombre permettant d'identifier le terrain, puis de l'extension de fichier "dat".

Pour utiliser la sauvegarde des terrains, nous allons ajouter un attribut mTerrainCreated à la classe PremiereApplication qui permettra de savoir si l'on a généré le terrain à partir d'une image ou bien si l'on a lu un fichier terrain. Dans le premier cas, on saura qu'à la fin de la méthode createTerrain() il faut penser à sauvegarder les fichiers de terrain pour le prochain lancement de l'application.

```
bool mTerrainCreated;
```

Initialisez sa valeur à false au début de la méthode createTerrain().

Maintenant, dans notre méthode definirTerrain(), il faut vérifier si le fichier terrain existe déjà ou bien s'il faut faire la génération depuis le heightmap comme le faisait jusqu'alors. Dans le second cas, on passe la variable mTerrainCreated à true.

```
void PremiereApplication :: definirTerrain (int x, int y)
{
    if (Ogre::ResourceGroupManager :: getSingleton () .resourceExists (mTerrainGroup->getResourceGroup (), mTerrainGroup->generateFilename (x, y)))
    {
        mTerrainGroup->defineTerrain (x, y);
    }
    else
    {
        Ogre::Image img;
        img.load ("terrain.png", Ogre::ResourceGroupManager :: DEFAULT_RESOURCE_GROUP_NAME);

        if (x % 2 != 0)
```

```

    img . flipAroundY () ;

    if (y % 2 != 0)
        img . flipAroundX () ;

    mTerrainGroup->defineTerrain (x, y, &img) ;
    mTerrainCreated = true ;
}
}

```

Revenons sur la condition à tester pour vérifier l'existence du fichier généré.

Grâce au Ogre::ResourceGroupManager, on peut vérifier s'il existe une ressource précise dans l'ensemble des ressources chargées au démarrage du programme. Les paramètres de la méthode sont le groupe de ressources dans lequel on veut chercher la ressource ainsi que le nom du fichier recherché. Vous voyez que le nom du fichier généré par le groupe de terrains dépend de ses coordonnées X et Y, ainsi que de la convention que l'on a définie au début.

Si le fichier est trouvé, on appelle la méthode defineTerrain() avec seulement les coordonnées en paramètres. Dans ce cas, Ogre va aller chercher directement le fichier correspondant à ces coordonnées. Dans le cas contraire, on exécute le bloc que l'on avait précédemment et qui charge le terrain à partir de l'image de heightmap.

Il ne reste plus qu'à demander la sauvegarde des fichiers si l'on a généré les terrains juste avant de libérer les ressources dans la méthode createTerrain() :

```

if (mTerrainCreated)
    mTerrainGroup->saveAllTerrains (true) ;

```

Lancez l'application, le temps de chargement doit être un peu plus long qu'auparavant car l'ordinateur sauvegarde en même temps les fichiers générés sur le disque. Une fois que l'application est lancée, fermez-la puis relancez-la. Le temps de chargement doit normalement être meilleur.

Vous devriez trouver les fichiers générés dans le dossier OgreSDK.media. Pour information, les miens font chacun une taille de 12 Mo.

Index

ALIGN_X_Z, 100
alignement du terrain, 100
Batch Size, 82
Blend Map, 85
 dirty(), 86
 update(), 86
bump mapping, 80
calque, 82
caméra
 distance de vue à l'infini, 78
createTerrain(), 99
defineTerrain(), 101
dirty(), 86
Filtrage anisotrope, 83
fusion de calques, 85
getDefaultImportSettings(), 100
getWidth(), 100
groupe de terrains, 99
heightmaps, 80, 101
 getWidth(), 100
ImportData, 81
inputScale, 82
load(), 100
loadAllTerrains(), 100
LOD, 82
matériaux, 83
maxBatchSize, 82
minBatchSize, 82
niveau de détail, 82, 83
niveau de filtrage, 83
 anisotrope, 83
 bilinéaire, 83
 trilinéaire, 83
noeud, 19
OGRE_DELETE, 79
OGRE_NEW, 79
prepare(), 81
setFilenameConvention(), 100
setMaxPixelError(), 79
setOrigin(), 100
Terrain
 ALIGN_X_Z, 100
 Batch, 82
 calque, 82
 Filtrage
 anisotrope, 83
 fusion de calques, 85
 ImportData, 81
 inputScale, 82
 maxBatchSize, 82
 minBatchSize, 82
 niveau de détail, 83
 niveau de filtrage, 83
 prepare(), 81
 TerrainGlobalOptions, 78
 TerrainGroup, 99
 defineTerrain(), 101
 getDefalutImportSettings(), 100
 setFilenameConvention(), 100
 setOrigin(), 100
 TerrainLayerBlendMap, 85
 texture, 82
update(), 86
worldSize, 82